

Ein mikrocontrollerbasiertes Programmiergerät für EPROMs und EEPROMs

Studienarbeit
im Studiengang Informatik

vorgelegt von

Volker Klasen	Jürgen Starek
204110016	204110559

Betreuer: Dr. Merten Joost, Institut für integrierte Naturwissenschaften, Abteilung Physik, Fachbereich 3: Naturwissenschaften

Koblenz, im Juli 2008

Erklärung
Wir versichern, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Volker Klasen Jürgen Starek

Inhaltsverzeichnis

1 Projektüberblick

Das Hardwarepraktikum der Informatikstudiengänge an der Universität Koblenz, das für den Studiengang Informatik eine Pflichtveranstaltung ist, soll die theoretischen Kenntnisse aus begleitenden Vorlesungen praktisch vertiefen. Neben grundlegenden Schaltungen und Schaltnetzen der Elektrotechnik sowie weiterführendem Programmieren von Mikrocontrollern ist der Aufbau von Schaltwerken mit Hilfe von Festwertspeichern (engl. Read-Only-Memory, ROM) ein Thema des Praktikums.

Die Programmierung der Festwertspeicher wird derzeit mit Hilfe von Einplatinencomputern des Typs Siemens ECB85 durchgeführt. Diese Experimentiercomputer verfügen mit einer hexadezimalen Tastatur und einer achtstelligen Sieben-Segment-Anzeige über eine einfache, wenig komfortable Benutzerschnittstelle. Ein fest gespeichertes Programm erlaubt das Schreiben kleiner Programme von unter 1 KiB in EPROMs. Obwohl die eingesetzten EPROMs eine Speicherkapazität von 2 KiB besitzen, ist die beschränkte Programmierfähigkeit kein Problem für den Einsatz im Hardwarepraktikum, weil die zu brennenden Programme in ihrer Größe bei weitem nicht an die Grenze von 1 KiB heranreichen.

Das Alter der ECB85, die im Jahre 1981 eingeführt wurden, führt allerdings mittlerweile zu häufigen Defekten. Ersatzteile für die Geräte sind teils nur noch schwer verfügbar.

Als Ersatz für die betagten Einplatinencomputer wurde jetzt ein benutzerfreundlicheres Programmiergerät gesucht, das ohne einen angeschlossenen PC verwendbar sein sollte. Das Gerät musste also eine Möglichkeit bieten, Daten in die PROMs zu schreiben und wieder auszulesen, aber auch die üblicherweise in PC-Software realisierten Funktionen für Eingabe und Bearbeitung der Daten anbieten. Daneben sollte es in der Lage sein, die gesamte Kapazität der EPROMs von 2 KiB ohne Einschränkungen zu nutzen.

Um eine kostengünstige und flexible Lösung zu erhalten, sollte eine Eigenentwicklung verwendet werden. Da in den Koblenzer Hardwarepraktika nur EPROMs der Serie 2716 und EEPROMs der Serie 2816 verwendet werden, konnte eine verhältnismäßig einfache Schaltung verwendet werden, so dass ein Aufbau mit den vorhandenen Hilfsmitteln denkbar schien: Mit Hilfe der freien Version der Layoutsoftware EAGLE von CadSoft [1] lassen sich zweilagige Platinen im halben Europakartenformat entwickeln, die sich mit der vorhandenen Ätzanlage und üblichen Werkzeugen gut fertigen und bestücken lassen.

Im Rahmen dieser Studienarbeit wurden die Hardware und Software eines solchen Programmiergeräts entwickelt.

2 EPROMs und EEPROMs

Unter der Bezeichnung Programmable Read Only Memories, kurz PROM, werden elektronische Speicherbausteine zusammengefasst, die sich mit beliebigen Daten beschreiben und danach immer wieder auslesen lassen. Das Beschreiben wird dabei üblicherweise als „brennen“ bezeichnet. PROMs sind frei addressierbar. Sie werden häufig als Speicher für Steuerprogramme genutzt, die sich zur Laufzeit nicht verändern.

[1] http://www.cadsoft.de

Im Rahmen des Hardwarepraktikums werden PROMs als Ersatz für größere Schaltnetze verwendet, die sonst für die in den Experimenten aufzubauenden Schaltwerke vonnöten wären. Die Festwertspeicher sind löschbar, um eine Korrektur der gespeicherten Daten zu erleichtern und zu verhindern, dass für jede Veranstaltung neue Speicherchips bereitgestellt werden müssen. Nach der zum Löschen verwendeten Technik unterscheidet man dabei Erasable PROMs (EPROMs) und Electrically Erasable PROMs (EEPROMs, selten auch E^2PROMs abgekürzt). Zum Löschen des Speicherinhalts werden EPROMs in geeigneten Löschgeräten mit UV-Licht bestrahlt; EEPROMs werden einfach durch erneutes Beschreiben mit einem konstanten Wert, etwa 0xFF, „gelöscht“. Bei EEPROMs ist ein Löschvorgang eigentlich nicht nötig, da bestehende Programme einfach überschrieben werden können; es ist jedoch bei der manuellen Fehlersuche übersichtlicher, wenn in den Speicherzellen hinter dem letzten Byte des Programms nur noch konstante Werte folgen.

Derzeit werden Speicher mit einer Kapazität von 2 KiB verwendet, genauer EPROMs vom Typ 2716 und EEPROMs des Typs 2816, die pinkompatibel zum Typ 2716 sind. Die Speicherchips haben ein 24-poliges DIL-Gehäuse, so dass sie leicht handhabbar sind. Ein UV-Löschgerät für das Praktikum ist bereits vorhanden, so dass sich das vorgestellte Programmiergerät auf das Löschen von EEPROMs beschränken kann.

3 Marktübersicht

3.1 Fertiggeräte

Im kommerziellen Bereich sind EPROM-Programmer am weitesten verbreitet, die nur aus der Schreibelektronik und einer IC-Fassung bestehen und über einen angeschlossenen PC bedient werden müssen. Geräte, die, wie gefordert, ohne PC auskommen und komplett eigenständig arbeiten können, sind selten und relativ teuer. Beispiele für solche Geräte sind das STAG P301[2], das derzeit für ca. 750 € erhältlich ist, der Shooter-XP[3] der Firma Logical Devices für ca. 500 $ oder der Dataman S4[4] von Duncan Instruments, der schon seit 2004 nicht mehr gebaut wird, aber gebraucht noch für ca. 600 € gehandelt wird.

Kommerzielle Geräte können in der Regel eine große Zahl verschiedener EPROMs beschreiben. Für den Einsatz im Praktikum ist das nicht erforderlich, so dass selbstentwickelte Geräte relativ einfach gehalten werden können. Das vorgestellte Gerät lässt sich mit Komponenten im Wert von etwa 30 € in wenigen Stunden aufbauen.

3.2 Komponenten

Das Angebot an Komponenten für einen Selbstbau ist derzeit sehr gut. Zur Steuerung des Geräts stehen verschiedene Familien von Mikrocontrollern zur Verfügung. Wegen der vor Ort bereits vorhandenen Programmiertechnik haben wir uns für die weit verbreiteten und günstigen Atmel-Mikrocontroller der Serien ATmega und ATtiny entschieden. Nachdem die Kontrollaufgaben nicht zeitkritisch und das eigentliche Programm relativ

[2] http://www.stag.co.uk/products_p301.html
[3] http://www.logicaldevices.com/Products/chipkopier.htm
[4] http://www.duncaninstr.com/datamns4.htm

einfach sind, wären auch leistungsschwächere, energieeffizientere Controller wie die der MSP430-Serie von Texas Instruments eine Alternative gewesen. Es war aber für einen wartungsfreundlichen Aufbau auch gewünscht, Controller im klassischen DIL-Gehäuse zu nutzen, die für einen leichten Austausch in IC-Sockeln montiert werden. Controller anderer Serien sind in dieser Bauform meist nicht mehr erhältlich, was ein weiterer Grund für die Nutzung der Atmel-Controller war.

Die zuvor verwendeten Siemens-Einplatinencomputer verfügten nur über ein einzeiliges LED-Display mit 8 Stellen. Verschiedene Menüebenen oder Befehle wurden mit Kennziffern angezeigt, die optisch nicht von den einzugebenden Daten abgesetzt waren. Um den Bedienkomfort zu verbessern, sollte im Nachfolgegerät auf jeden Fall ein mehrzeiliges Display genutzt werden, um eine Menüführung mit Klartextbezeichnern und eine übersichtliche, mehrzeilige Programmdarstellung zu ermöglichen. Hier werden derzeit sehr viele Bauformen angeboten. Sowohl grafikfähige als auch textorientierte LC-Displays, die vier oder mehr Zeilen von mehr als 20 Zeichen (der für die Darstellung im Stil eines Hexeditors notwendigen Zeilenlänge) darstellen können, kosten aber zwischen 30 € und 50 €, was selbst für eine Kleinstserie zu teuer schien. Andere Technologien wie OLEDs oder TFTs sind noch wesentlich teurer. Beschafft wurden schließlich einige vierzeilige Displays, die mit einer Zeilenlänge von 27 Zeichen genug Platz für die geplante Bedienoberfläche boten und als Restposten günstig erhältlich waren.

Passende Tastaturen (es werden 16 Tasten für die Eingabe der im Hexadezimalsystem geschriebenen Programme benötigt, dazu kommen noch mindestens neun Steuertasten) waren nicht kommerziell erhältlich, so dass hier der aufwändige, aber preisgünstige Aufbau aus Einzeltasten gewählt wurde. Leider lässt die Ergonomie einer solchen Tastatur deutlich zu wünschen übrig, was aber zumindest beim Prototyp in Kauf genommen wurde: In aller Regel wird man auf der Tastatur keine allzu langen Programme eingeben müssen.

4 Entwurf

Vor dem Entwurf der Schaltung musste ein geeigneter Mikrocontroller gefunden werden, der über für das Projekt ausreichende Rechenleistung, Speicherkapazität und I/O-Anschlüsse verfügen musste. Es zeigte sich, dass aktuelle Controller durchweg über genügend Rechenleistung verfügen und dieses Kriterium für die Auswahl eines Controllers nicht entscheidend sein würde.

Für die Versuche im Hardwarepraktikum sollten EPROMs des Typs 2716 mit 2048 Byte Speicherkapazität bzw. voll pinkompatible EEPROMs der Serie 2816 verwendet werden. Um die Daten für diese Bausteine komplett zwischenspeichern zu können, muss der verwendete Controller also über mindestens 2 KiB Speicherkapazität verfügen.

Diese Speicherbausteine sind über 11 Adressleitungen und 8 Datenleitungen ansprechbar. Zum Beschreiben der Bausteine sind daneben noch drei weitere Leitungen mit Steuerfunktionen nötig (vgl. 5.1.1, S. 10), so dass allein für die Ansteuerung der PROMs 22 I/O-Leitungen belegt werden.

Die Eingabe von Daten für PROMs erfolgt üblicherweise in hexadezimaler Schreibweise. Entsprechend ist mindestens eine Eingabetastatur mit 16 Tasten notwendig. Beim

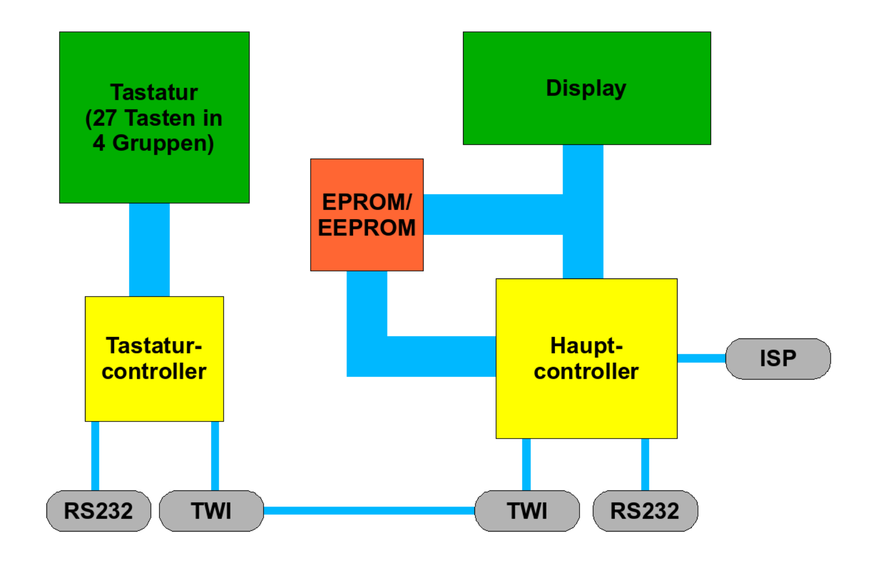


Abbildung 1: Konzept des Programmiergeräts

vorliegenden Bedienkonzept mit Menüsteuerung kommen dazu noch vier Cursortasten, vier Softkeys, die je nach ausgeführtem Programm unterschiedlich belegt sind, sowie je eine Bild-auf-, Bild-ab- und Bestätigungstaste. Die Tastatur ist beim aufgebauten Prototyp so angeordnet, dass die Softkeys unmittelbar unter der Unterseite des Displays liegen. Auf diese Anordnung ist auch die Software abgestimmt, die die Beschriftung der Softkeys in der untersten Displayzeile anzeigt. Auf den oberen drei Zeilen werden Statusinformationen oder die Daten des PROMs in einer Editoransicht angezeigt.

Die eingesetzten LC-Displays basieren auf zwei HD44780-Controllern und werden über 8 Datenleitungen und 4 Steuerleitungen angesprochen. Zusammen mit den 22 Leitungen zum EPROM und zwei Leitungen für eine RS232-PC-Schnittstelle sowie einer Leitung für die Abfrage des EPROM-/EEPROM-Umschalters musste der Mikrocontroller also über mindestens 37 I/O-Leitungen verfügen.

Für eine Hexadezimaltastatur mit den benötigten Sondertasten, die in Form einer Matrix verkabelt wird, sind zusätzlich 4 Gruppen- und 8 Zeilenleitungen notwendig, insgesamt wären das also 49 Anschlüsse. Controller mit mehr als 32 I/O-Pins (4 Ports) sind jedoch nur in SMD-Bauform erhältlich. Um beim Aufbau auf Controller im wartungsfreundlicheren DIL-Gehäuse zurückgreifen zu können, mussten Möglichkeiten gesucht werden, I/O-Leitungen einzusparen.

Die Verwendung eines zweiten Controllers, der nur für die Vorverarbeitung von Tastendrücken zuständig sein sollte, machte es möglich, die Tastendrücke über ein Zweidrahtbussystem zu übertragen und so 10 Leitungen einzusparen.

Um weitere Leitungen einzusparen, wurden die Datenleitungen von EPROM und Display zu einem Bus zusammengelegt. Dadurch kann nicht gleichzeitig auf EPROM und Display zugegriffen werden, was aber durch den Zwischenspeicher in den Displaycontrol-

lern und die Art des Schreibprozesses der PROMS, bei dem das zu schreibende Datum nicht permanent am Bus anstehen muss, auch nicht nötig ist. Somit werden nur noch 31 I/O-Pins benötigt.

Unter den Atmel-Controllern, die diese Anforderungen erfüllen, war der ATmega644 der einzige, der im DIL-Gehäuse erhältlich war. Er bietet 32 I/O-Leitungen und 4 KiB internen Speicher, so dass neben einem Editorpuffer, der sämtliche im EPROM abgelegten Daten aufnehmen kann, noch genügend Platz für Programmvariablen bleibt. Da der ATmega644 anfangs noch nicht in kleinen Stückzahlen erhältlich war, musste die Entwicklung der Schaltung mit dem pinkompatiblen ATmega16 erfolgen, der ebenso wie die erforderliche Programmiertechnik bereits im Labor vorhanden war, aber mit 1 KiB RAM über zu wenig Arbeitsspeicher verfügt, um einen hinreichend großen Editorpuffer anlegen zu können. Übergangsweise wurde bei diesem Controller ein Editorpuffer von nur 512 Byte Größe verwendet.

Die Verarbeitung der Tastatureingaben übernimmt ein ATtiny2313. Dieser Controller liest ständig den Zustand der Tastatur aus und meldet ihn über den TWI-Bus an den Hauptcontroller. Diese Schnittstelle ist im Wesentlichen identisch zum bekannten I^2C-Bus von Philips, dessen Name aus lizenzrechtlichen Gründen von Atmel nicht übernommen wurde[5]. Ein Vorteil der Ausführung mit zwei Controllern ist, dass Tastendrücke in jedem Betriebszustand zuverlässig aufgezeichnet werden: Selbst wenn der Hauptcontroller beschäftigt ist, kann der Tastaturcontroller Tastendrücke puffern und zur Verarbeitung an den Hauptcontroller weitergeben, sobald dieser wieder bereit ist.

5 Aufbau

Die Hardware des Programmiergeräts ist auf drei Platinen verteilt, dazu kommen die Tastatur und das Display.

Die Hauptplatine integriert den Hauptcontroller mit seiner externen Beschaltung, einen step-up-Spannungsregler für die Erzeugung der Brennspannung und einen Pegelwandler für die RS232-Schnittstelle. Steckerleisten stellen die Signale für den Anschluss des EPROM-Sockels und des LC-Displays zur Verfügung, und über serielle Schnittstellen ist der Anschluss des Tastaturcontrollers, eines ISP-Programmiergeräts für den Mikrocontroller und eines PCs möglich.

Neben der Hauptplatine entstand eine kleine Platine für den abgesetzten Tastaturcontroller, die nur eine Fassung für einen ATtiny2313 und Anschlüsse für die Tastatur, Stromversorgung, den TWI-Bus und zwei Datenleitungen einer RS232-Schnittstelle bietet. Diese Schnittstelle ist für die Ausgabe von Debugging-Daten vorbereitet und müsste, wenn sie genutzt werden soll, noch mit einem MAX232-Treiber-IC verbunden werden.

Bei den Versionen 0.31 und 0.32 ist darüber hinaus noch eine Platine für einen ZIF-Sockel (Zero Input Force, Nullkraft-Sockel), der als Fassung für die PROMs dient, nötig.

Die Schaltpläne und Platinenlayouts der jeweils letzten Platinenversion sind im Anhang wiedergegeben. Sie wurden mit der Layoutsoftware Eagle von CadSoft [6] erstellt, die

[5] http://www.nongnu.org/avr-libc/user-manual/group__twi__demo.html
[6] http://www.cadsoft.de

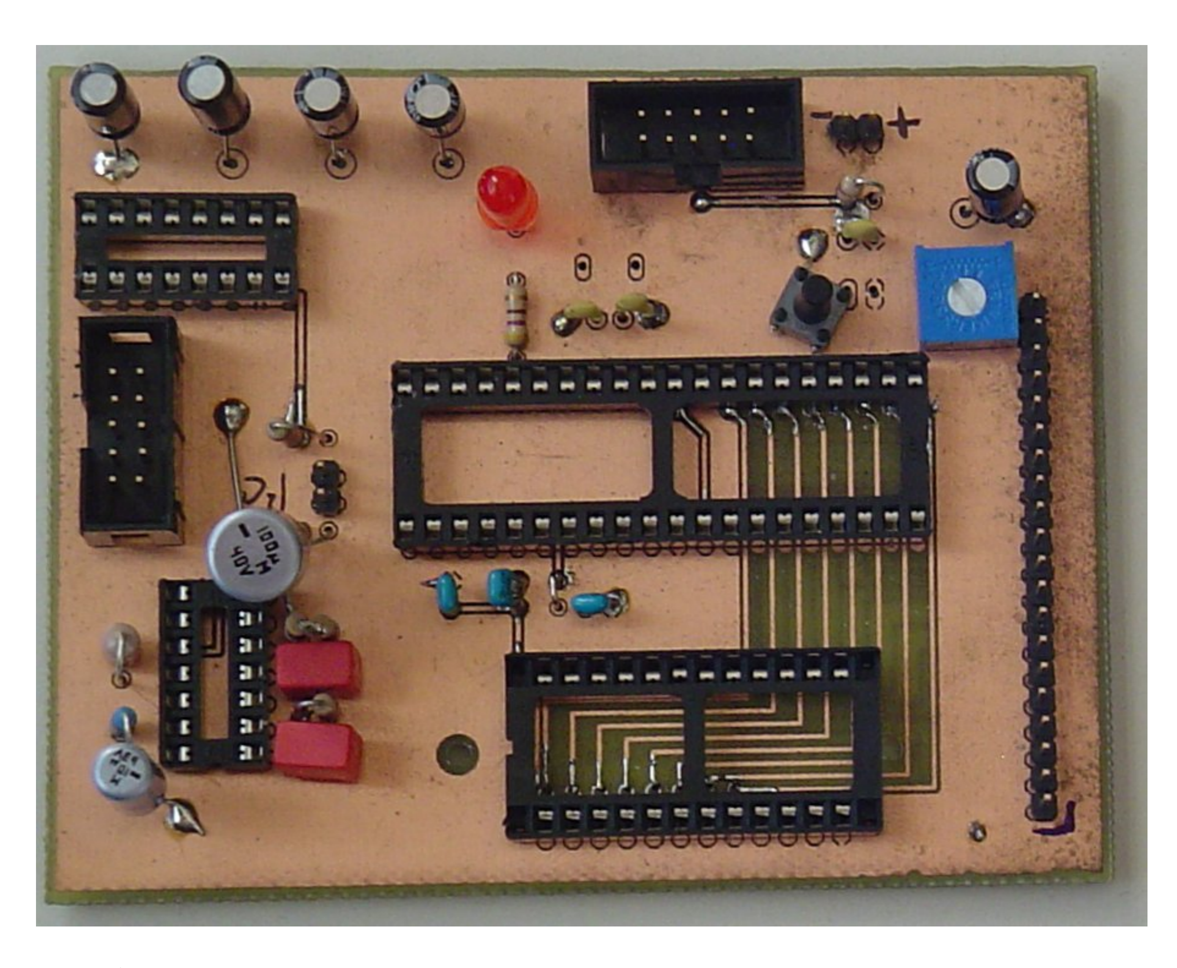


Abbildung 2: Die Platinenversion 0.2, die nur EPROMs verarbeiten konnte.

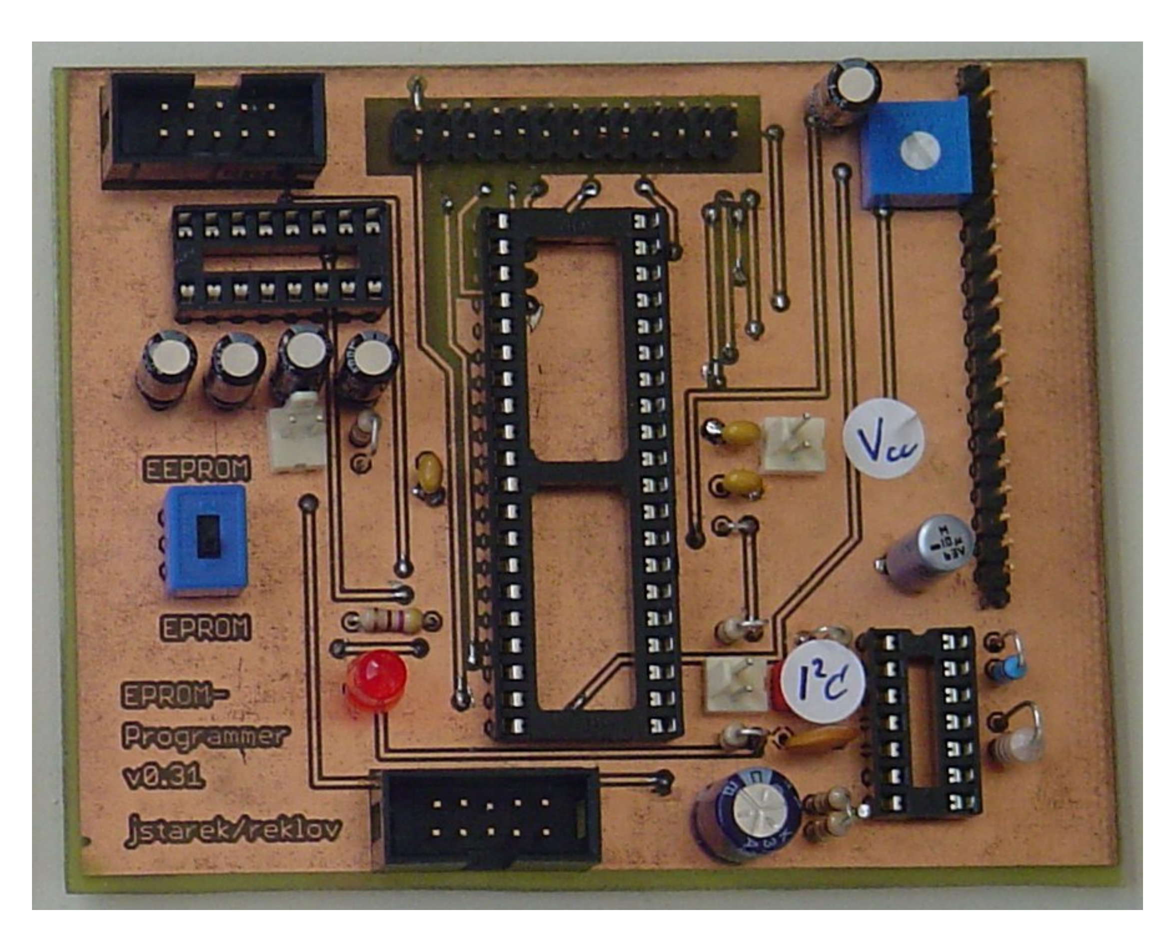


Abbildung 3: Die Platinenversion 0.31, mit der eine Umschaltung zwischen EPROM- und EEPROM-Betrieb integriert wurde und die für einen abgesetzten PROM-Sockel ausgelegt ist.

Version	Beschreibung
0.1	Erster Entwurf, nicht umgesetzt
0.2	Erster aufgebauter Protoyp, nur für EPROM-Betrieb geeignet
0.3	EPROM-/EEPROM-Umschaltung hinzugefügt
0.31	Zweiter aufgebauter Prototyp mit kleinen Korrekturen
0.32	Vereinfachte Masseverbindung für leichtere Bestückung

Tabelle 1: Entwurfsversionen der Hauptplatine

Platinen wurden selbst geätzt. Insgesamt entstanden fünf Versionen der Hauptplatine.

Alle Platinenlayouts sind für die manuelle Bestückung mit bedrahteten Bauelementen ausgelegt, um die Montage möglichst einfach zu gestalten. Für alle ICs sind DIL-Gehäuse vorgesehen; sie sollten beim Aufbau grundsätzlich mit Sockeln montiert werden, um so einen Wechsel zu erleichtern. Auf SMD-Technik wurde komplett verzichtet.

Da es schwer ist, im Handel Nullkraftanschlüsse für die an den Displays angebrachten Folienleiter in kleinen Stückzahlen zu bekommen, wurden diese abgelötet und durch handelsübliche Flachbandkabel mit 1,27 mm-Raster ersetzt. Die Anschlüsse auf der Platine sind für entsprechende Pfostenstecker ausgelegt.

Die ältere Version 0.2 war für eine offene Montage vorgesehen, so dass der PROM-Sockel direkt auf der Platine integriert war. Die Versionen 0.31 und 0.32 sind darauf ausgelegt, in einem Gehäuse mit abgesetztem PROM-Sockel montiert zu werden. Um diese Art der Montage zu erleichtern, sind alle Steckverbinder an den Rand der Platine verschoben worden. Da es bei der manuellen Bestückung der zweiseitigen Platinen nicht immer einfach ist, Durchführungspins auf beiden Seiten der Platine zu verlöten, wurde in der neuesten Version 0.32 noch eine zusätzliche Verbindung zweier Masseflächen eingefügt.

5.1 Hauptplatine

5.1.1 Beschaltung des Hauptcontrollers

Der ATmega644 benötigt nur eine sehr einfache externe Beschaltung. 32 der 40 Pins des DIL-Gehäuses stehen als I/O-Pins zur Verfügung.

Der Mikrocontroller wird, wie im Datenblatt vorgeschlagen, über die Pins 9 bis 11 und 30 bis 32 mit Versorgungs- und Referenzspannung versorgt. Der Reset-Anschluss, Pin 9, liegt über den Pull-Up-Widerstand R1 an der Versorgungsspannung und kann bei Bedarf über JP3 auf Masse gelegt werden, was einen Reset des Controllers auslöst. Nachdem wir für unsere Anwendungen keine besonderen Anforderungen an die Konstanz der Taktfrequenz stellen, bleiben die Anschlüsse für einen externen Quarz, Pin 12 und 13, unbeschaltet. Damit wird der interne Taktgenerator des Controllers mit einer Frequenz von 1 MHz benutzt.

Die I/O-Pins des Controllers werden zu vier Ports, bezeichnet mit A bis D, zusammengefasst.

Unsere Schaltung nutzt Port A, also die Pins PA0 bis PA7, als kombinierten Datenbus für das PROM und das LC-Display. Die hier angelegten 8 Bit breiten Datenworte werden

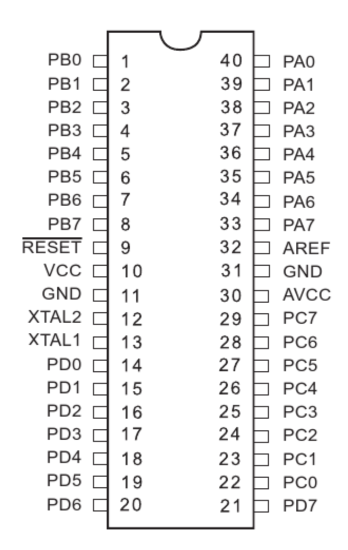


Abbildung 4: Pinbelegung des ATmega644. Die Pins von Port A sind mit PA0 bis PA7 bezeichnet, die der anderen Ports entsprechend. Die übrigen 8 Pins werden durch die externe Beschaltung belegt und stehen nicht für Ein- und Ausgabeleitungen zur Verfügung.

je nach Betriebszustand von den eingebauten Controllern des Displays ausgelesen, in ein PROM geschrieben oder dienen zur Übertragung der aus einem PROM ausgelesenen Daten. Dabei wird der Bus nicht zwischen PROM-Sockel und Display umgeschaltet, sondern die Werte stehen immer am gesamten Bus an. Ob und wo sie ausgelesen werden, hängt von der Ansteuerung des Displays bzw. des PROMs ab: Über die Funktion `misc_set_data_direction()` aus der `misc.c` wird bestimmt, welcher der drei Bus-Teilnehmer (Hauptcontroller, Display oder PROM) auf den Bus schreiben darf. Beim Hauptcontroller wird dazu das Register DDRA entsprechend gesetzt. Damit das Display auf den Bus schreiben kann, muss die Leitung R/W auf 1 gesetzt werden, andernfalls liest es vom Bus. Wird beim PROM eine 1 auf die Leitung $\overline{\text{G}}$ gelegt, so wird der Ausgang des PROMs gesperrt, sonst liegen die Daten des PROMs auf dem Bus.

Die Funktion `misc_set_data_direction()`, die als Argument den Teilnehmer erhält, der auf den Bus schreiben soll, stellt dabei zunächst sicher, dass keiner der anderen Teilnehmer mehr darauf schreiben darf. Anschließend erlaubt sie Ersterem, auf den Bus zu schreiben (vgl. `misc.c`, S. 35 ff.).

Aufgerufen wird diese Funktion von anderen Funktionen, die anschließend auf den Bus schreiben wollen, wie z.B. der Funktion `lcd_write()` aus der `lcd.c` (s.S. 39 ff.) oder der ähnlich aufgebauten Funktion `eprom_write()` aus der Datei `eprom.c` (s.S. 47 ff.).

Port B dient im Normalbetrieb als Anschluss für einen 8 Bit breiten Bus, an dem die niederwertigen Bits der 11 Bit breiten, im PROM anzusprechenden Adresse anliegen (die höherwertigen Bits liegen auf den Pins 4 bis 6 von Port D). Die drei Pins PB5 bis PB7 dienen gleichzeitig zur Verbindung mit der auf einen zehnpoligen Wannenstecker

herausgeführten ISP-Schnittstelle[7].

Den Port C teilen sich einige Steuerleitungen: Über PC0 und PC1 läuft der TWI-Bus, der den Hauptcontroller mit dem Tastaturcontroller verbindet. Diese beiden Pins sind über Pull-Up-Widerstände von 10 kΩ an die Versorgungsspannung gelegt, so dass an ihnen immer ein definiertes Potential anliegt. Die folgenden vier Pins sind mit Kontrollfunktionen für das Display belegt. PC6 schaltet die Ausgänge des EPROMs zwischen hoch- und niederohmig um; sie sind hochohmig, wenn an PC6 ein High-Pegel liegt. PC7 steuert den eigentlichen Brennvorgang: Liegt hier ein High-Pegel, wird auf das PROM geschrieben.

Port D stellt mit den Pins PD0 und PD1 eine serielle Schnittstelle als Anschlussmöglichkeit für Computer bereit. Der nächste Pin von Port D, PD2, dient als Anschluss für eine LED. Diese LED wird von unserem Programm zur Anzeige von Fehlerzuständen benutzt und kann in Fehlerfällen hilfreich sein, in denen das LC-Display nicht mehr angesprochen werden kann. PD3 ist für die Umschaltung zwischen dem EPROM- und dem EEPROM-Modus zuständig; die Software wertet aus, ob über den Schalter S1 an diesen Pin Masse- oder V_{CC}-Pegel gelegt wurde. Auf den folgenden drei Pins PD4 bis PD6 liegen die höherwertigen Bits der 11 Bit breiten PROM-Adresse (s.o.). PD7 schließlich kontrolliert die Brennspannung: Im EEPROM-Modus wird er über S1 an Pin 8 des Prom-Sockels gelegt, so dass dort der Brennimpuls mit 0 V direkt angelegt werden kann. Im EPROM-Modus wird er als Steuerleitung für den step-up-Regler verwendet, der 5 V oder 25 V (Brennspannung) an das EPROM liefert.

5.1.2 Erzeugen der Brennspannung

Da die Schaltung mit 5 V Betriebsspannung arbeitet, muss für die Erzeugung der Brennspannung von 25 V ein gewisser Aufwand getrieben werden. Grundlage dafür ist ein Schaltregler-IC vom Typ TL 497 von Texas Instruments [TI95]. Dieser IC enthält die Grundschaltung eines Aufwärts-Spannungswandlers. Bei diesen Wandlern wird ein Ausgangskondensator relativ hoher Kapazität, in unserer Schaltung C6 mit 100 μF, auf eine Spannung aufgeladen, die deutlich über der Betriebsspannung der Schaltung liegt. Dazu wird eine Induktivität, hier L2, wechselweise an die Betriebsspannung gelegt und dann mit dem Ausgangskondensator verbunden. In der Spule wird also ein Magnetfeld aufgebaut. Beim Umschalten bricht es zusammen und induziert dabei eine Spannung in der Spule. Die zuvor im Magnetfeld gespeicherte Energie wird dabei in den Ausgangskondensator übertragen, der sich so auf immer höhere Spannungen auflädt.

Betreibt man eine solche Schaltung ohne weitere Regelung, ist sie weder kurzschlussfest (da keine Bauteile zwischen Ein- und Ausgang liegen, die für Gleichspannungen einen nennenswerten Widerstand aufweisen) noch leerlauffest: Die Spannung im Ausgangskondensator steigt an, bis er durchschlägt. Um das zu verhindern, wird beim TL497C eine Regelungsschaltung verwendet. Die beiden Widerstände R5 und R6 wirken, zwischen Ausgangsspannung und Masse geschaltet, als Spannungsteiler. Der IC überwacht die

[7] ISP: In System Programming. ISP ermöglicht es, Software in Controller zu laden, ohne sie dafür aus der Schaltung ausbauen zu müssen. Für die Atmel-Controller existieren Programmieradapter, die an SV1 angeschlossen werden können und die Verbindung zwischen Schaltung und dem PC mit der Programmiersoftware herstellen.

Spannung an Pin 1, der zwischen den beiden Widerständen liegt. Ist sie kleiner als 1,2 V, wird der interne Oszillator eingeschaltet und damit Energie von der Spule auf den Ausgangskondensator umgeladen, die Ausgangsspannung also erhöht. Wird diese Feedbackspannung daraufhin größer als 1,2 V, wird der Oszillator dann wieder abgeschaltet, was das Erzeugen einer zu hohen Spannung am Ausgangskondensator verhindert.

Im Dauerbetrieb mit einer Last am Ausgang ist die Ausgangsspannung nur von der zeitlichen Abfolge der Umladevorgänge zwischen Spule und Kondensator abhängig ([KSW06], S. 496), weder von der Last noch von der Induktivität der Spule. Das Tastverhältnis wird in der vorliegenden Schaltung durch den Kondensator C11 festgelegt. Er wird durch eine Konstantstromquelle geladen. So lange eine bestimmte Spannungsschwelle noch nicht erreicht ist, schaltet der interne Oszillator die Spule an die Versorgungsspannung. Beim Erreichen der Spannungsschwelle wird der Timerkondensator entladen und die Spule an den Ausgangskondensator geschaltet, an den sie ihre Energie abgibt. Das Tastverhältnis ist dabei nur von der Kapazität des Timerkondensators abhängig, aber nicht von der Eingangsspannung.

Zusätzlich zu diesem Schaltungsteil, der den Ausgangskondensator vor Überspannungen schützt, enthält der IC noch eine Schutzschaltung gegen zu hohe Umladeströme. Wenn über einen zwischen Pin 13 und 14 geschalteten Widerstand, hier R4, eine höhere Spannung als 0,7 V abfällt, wird eine interne Strombegrenzung aktiviert, die verhindert, dass die Spule durch zu hohe Ströme beschädigt wird.

Diese Schaltung entspricht der Konfiguration, die schon bei den ECB-Geräten verwendet wurde.

5.1.3 RS232-Schnittstelle

Die RS232-Schnittstelle codiert logische Zustände binär als Spannungspegel von -25 V bis -3 V sowie 3 V bis 25 V. Der negative Spannungsbereich steht dabei für eine logische 1, der positive für eine logische 0. Zur Erzeugung dieser Spannungen laufen die Signale der Pins D0 und D1 des Hauptcontrollers über ein Pegelwandler-IC vom Typ Maxim MAX232. Der MAX232 verwendet intern eine Ladungspumpe, die die Betriebsspannung von 5 V verdoppelt, und einen Inverter, der zu den +10 V noch -10 V bereitstellt [MAX04]. Durch diese transparente Pegelwandlung kann der Hauptcontroller das Protokoll, das für die Kommunikation mit dem PC über diese Schnittstelle verwendet wird, unmittelbar über D0 und D1 abwickeln.

Die Erzeugung und Verarbeitung des Protokolls wird von den Funktionen in der `pc.c` (s.S. 77) erledigt.

5.2 Tastatur und Tastaturcontroller

Die Tasten der hexadezimalen Tastatur sind mit Zeilen- und Spaltenleitungen zu einer Matrix verkabelt, so dass jede Taste über den Schnittpunkt von Zeilen- und Spaltenleitung, an dem sie liegt, eindeutig identifizierbar ist. Diese Leitungen liegen direkt an den I/O-Pins des Controllers. Wegen der Funktion der Zeilenleitungen im Programm des Tastaturcontrollers (s.u.) werden wir sie in der Regel als Gruppenleitungen bezeichnen.

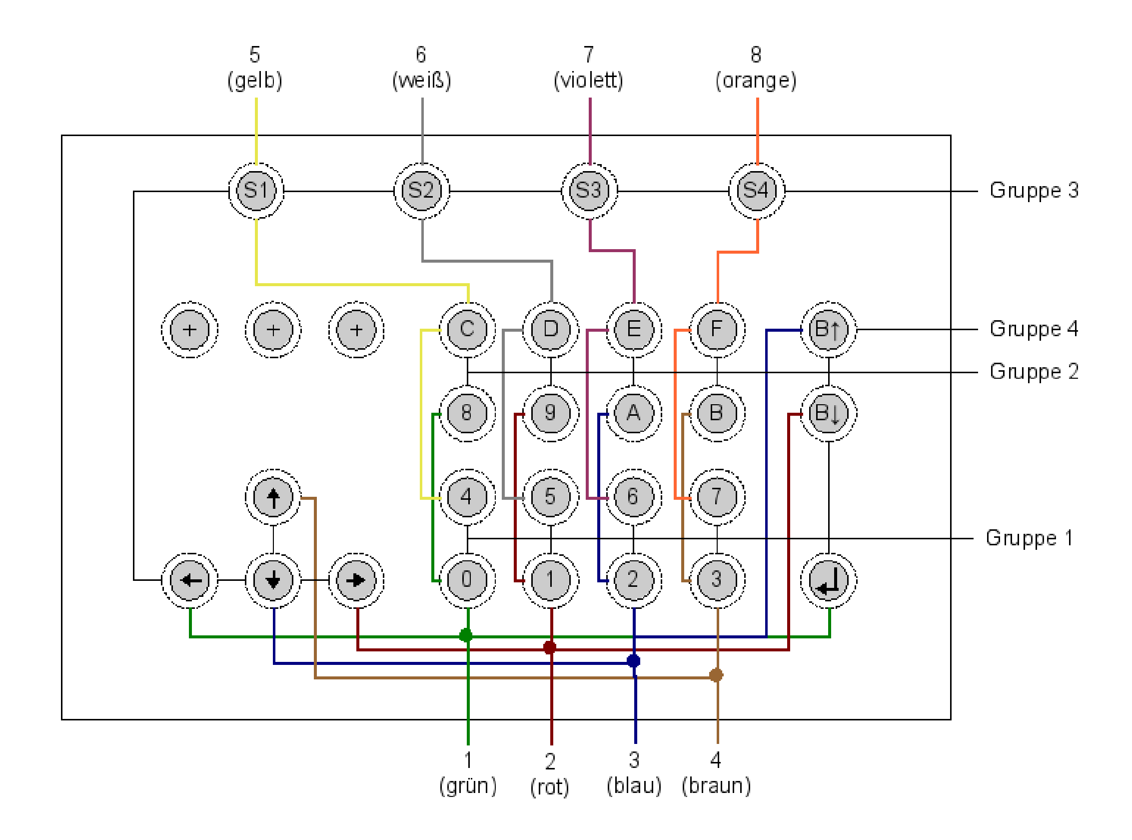


Abbildung 5: Skizze der Tastatur. Jede Taste ist mit einer der vier schwarzen Gruppenleitungen und einer der farbcodierten Spaltenleitungen verbunden. Die drei unbeschrifteten Bohrungen zwischen Pfeiltasten und Softkeys sind für Kontroll-LEDs vorgesehen.

	Gruppe 1	Gruppe 2	Gruppe 3	Gruppe 4
1 (grün)	0	8	Links	OK
2 (rot)	1	9	Rechts	PD
3 (blau)	2	A	Unten	PU
4 (braun)	3	B	Oben	
5 (gelb)	4	C	SK 1	
6 (weiß)	5	D	SK 2	
7 (violett)	6	E	SK 3	
8 (orange)	7	F	SK 4	

Tabelle 2: Verdrahtung der Tastaturmatrix. Die Tasten liegen am Kreuzungspunkt einer Gruppenleitung (schwarz) und einer der durchnummerierten Spaltenleitungen.

Der Schaltung der Platine des ATtiny2313, der für die Auswertung der Tastatureingaben zuständig ist, ist noch einmal deutlich einfacher als die der Hauptplatine. Der Controller ist über die Pins 10 und 20 unmittelbar an die Versorgungsspannung angeschlossen. Pin 1 kann über einen Jumper an Masse gelegt werden und löst damit einen Reset aus. Die Pins 2 und 3 sind für Diagnosezwecke als TX- und RX-Leitung einer RS232-Verbindung nutzbar; diese Funktion wird im Normalbetrieb aber nicht benötigt und ist daher im Programm des Controllers nicht implementiert. Dennoch ist die Nutzung dieser Schnittstelle auf der Platine vorbereitet: Beide Pins werden über eine Steckerleiste zugänglich gemacht. Um diese Diagnoseschnittstelle zu nutzen, müssten ihre Signale wie beim Hauptcontroller mit Hilfe eines Pegelwandler-Bausteins an die korrekten RS232-Pegel angepasst werden.

An den beiden folgenden Pins 4 und 5 könnte ein externer Quarzoszillator angeschlossen werden. Da wir aber, wie beim Hauptcontroller auch, hier keine besonderen Ansprüche an die Qualität der Taktfrequenz stellen und mit dem internen Taktgenerator arbeiten, dienen diese beiden Pins als Anschlüsse für Datenleitungen.

Auf den Pins 4 und 5 liegen zwei der Spaltenleitungen. Die folgenden vier Pins werden von den Gruppenleitungen belegt, die Pins 12 bis 16 und 18 von den übrigen Spaltenleitungen. Die beiden Leitungen des Zweidrahtbusses, der die Daten vom Tastatur- zum Hauptcontroller überträgt, belegen schließlich die Pins 17 und 19.

Die nicht ganz regelmäßige Verteilung der Datenleitungen auf die Anschlüsse wird zum Teil durch das dadurch einfachere Platinenlayout, zum Teil aber auch durch die Belegung der Pins des Mikrocontrollers erzwungen.

5.3 Display

Das verwendete LC-Display vom Typ C2704M kann 108 Zeichen in vier Zeilen zu je 27 Stellen darstellen. Jedes Zeichen ist als Punktmatrix aus 8 Zeilen mit je 5 Pixeln (die 8. Zeile ist normalerweise für den Cursor reserviert) ausgeführt, so dass neben alphanumerischen Zeichen auch viele Sonderzeichen dargestellt werden können. Der EPROM-Programmer nutzt allerdings nur den üblichen ASCII-Zeichensatz.

Gesteuert wird das Display von zwei Controllern, von denen einer den oberen beiden, der andere den unteren beiden Zeilen zugeordnet ist. Die Controller sind zum Befehlssatz

der verbreiteten HD44780-Controller kompatibel.

5.3.1 Pinbelegung

Bei den verwendeten Displays kommt eine häufig verwendete externe Beschaltung mit 8 Datenleitungen sowie verschiedenen Leitungen zur Spannungsversorgung und zur Ansteuerung der Controller zum Einsatz. Insgesamt kommen hier damit 15 Leitungen für die Ansteuerung des Displays zusammen, von denen 12 I/O-Ports am Hauptcontroller belegen.

Pin	Funktion	
1	GND	GND
2	V_{pp}	5 V
3	V_0	0-4 V
4	RS	Register Select
5	R/W	Read/Write
6	E1	Controller für obere Zeilen
7	E2	Controller für untere Zeilen
8	D0	Datenleitung
9	D1	Datenleitung
10	D2	Datenleitung
11	D3	Datenleitung
12	D4	Datenleitung
13	D5	Datenleitung
14	D6	Datenleitung
15	D7	Datenleitung

Tabelle 3: Pinbelegung des C2704M. Diese Belegung ist bei Displays mit zwei HD44780-kompatiblen Controllern sehr weit verbreitet.

Die Stromversorgung des Displays erfolgt über seine Anschluss-Pins 1 und 2. An Pin 3 liegt eine über das Potentiometer R7 einstellbare Spannung von 0 bis +4 V, die den Kontrast der Zeichen auf dem Display bestimmt.

HD44780-Controller unterscheiden auf dem Datenbus zwischen Befehlen zur Steuerung des Displays (Bildschirm löschen, Cursor ein- bzw. ausschalten usw.) und anzuzeigenden Daten. Ob ein am Bus anstehendes Datenwort als Befehl interpretiert werden soll oder nicht, wird über Pin 4 (Register Select, kurz RS) bestimmt: liegt hier ein Low-Pegel, wird das anstehende Datum ins Befehlsregister des aktiven Controllers (s.u.) übernommen und entsprechend als Befehl interpretiert.

Mit Hilfe von Pin 5 (R/W) können die Daten, die in den Registern des Controllers stehen, wieder ausgelesen werden. Er schaltet den Controller in den Auslese-Modus, wenn hier ein High-Pegel ansteht.

Die beiden folgenden Pins 6 und 7 (E1 und E2) aktivieren jeweils den oberen bzw. unteren im Display integrierten Controller, so dass dieser den anstehenden Befehl (bzw. die anstehenden Daten) übernehmen kann.

Zuletzt folgen auf den Pins 8 bis 15 die Datenleitungen D0 bis D7, über die Nutzdaten ans Display übertragen werden. Der Displaycontroller kann sowohl mit 4 als auch mit 8 Bit breitem Datenbus arbeiten. Im 4-Bit-Modus werden die 8 Bit breiten Datenworte auf 2 aufeinanderfolgende 4-Bit-Nachrichten, die über die Leitungen D4 bis D7 übertragen werden, aufgeteilt; die anderen Datenleitungen werden dabei nicht benutzt.

5.3.2 Ansteuerung der 44780-kompatiblen Controller

Die Ansteuerung der beiden verbauten Displaycontroller erfolgt also über die Pins 4 bis 15. Eine Übersicht über die möglichen Befehle gibt Tabelle 4 (Quelle: [Hit44780]), auf die sich auch die im Folgenden genannten Befehlsnummern beziehen. Im Folgenden soll eine typische Möglichkeit zur Ansteuerung des Displays gezeigt werden.

In der Initialisierungsphase werden die Controller zunächst parametrisiert. Das erfolgt mit Befehl 6, der verschiedene Grundeinstellungen ermöglicht. Bit D4 wählt zwischen einem Betrieb mit 4 und 8 Bit breitem Bus aus, wobei eine 1 den 8-Bit-Betriebsmodus wählt. D3 teilt dem Controller mit, ob er nur eine Displayzeile (0) oder 2 bzw. 4 Displayzeilen in einem Kombidisplay wie dem hier verwendeten zu verwalten hat. D2 schaltet zwischen zwei Zeichengrößen um. Wird hier eine 0 geschickt, wird der Controller für den Betrieb mit 5×7 Pixeln konfiguriert, bei einer 1 für 5×10. Das letzte beachtete Bit, D1, schaltet bei einer gesetzten 1 einen bei manchen Displays notwendigen Spannungsinverter ein.

Für den 8-Bit-Betrieb bei unserem mehrzeiligem LC-Display wird also das Datum 0x38 bei RS=0 und R/W=0 in den Controller geschrieben. Danach wird der Cursor durch Schreiben von 0x02 auf die Startposition gesetzt.

Um Zeichen auf dem Display darzustellen, müssen die ASCII-Codes der darzustellenden Zeichen in den Speicher des Displaycontrollers geschrieben werden. Die HD44780-Controller unterscheiden zwischen zwei Speicherbereichen, dem sog. CG-RAM, in dem benutzerdefinierte Glyphen abgelegt werden, und dem sog. DD-RAM, in dem die anzuzeigenden Texte gespeichert werden. Zum Schreiben von Daten in diesen Speicher muss also zunächst der DD-RAM als Ziel des folgenden Datentransfers eingestellt werden. Dies geschieht über Befehl 8: es werden die Leitungen RS (Register Select) und R/W (Read/Write) auf 0 gesetzt, so dass die folgenden Daten als Befehl interpretiert werden und ins Steuerregister eingelesen werden, und die 7-bitige DD-RAM-Adresse zusammen mit einer 1 (für DD-RAM) am höchstwertigen Bit an die Datenleitungen gelegt. Anschließend wird der 8-Bit-Code für das darzustellende Zeichen mit RS=1 und R/W=0 in das DD-RAM geschrieben. Nach dem Schreibzugriff wird die DD-RAM-Adresse automatisch inkrementiert (bzw. dekrementiert), so dass fortlaufende Schreibzugriffe möglich sind.

Aber nicht alle Zeichen, die im DD-RAM liegen, werden auch dargestellt, denn die Controller unterstützen intern zwei Zeilen mit je 40 Zeichen Länge. Somit bietet das DD-RAM beim hier verwendeten Display mit vier Zeilen zu je 27 Zeichen Platz für mehr Zeichen, als angezeigt werden können. Um Hardwareherstellern zu ermöglichen, Displays mit kürzeren Zeilenlängen an diesen Controllern zu betreiben, bieten die Controller eine Funktion zum Verschieben des dargestellten Inhalts, ähnlich einem virtuellen Anzeigefenster, das über die Daten geschoben werden kann. Wie sich das Display in dieser Hinsicht

Nr.	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Beschreibung
1	0	0	0	0	0	0	0	0	0	1	Display löschen
2	0	0	0	0	0	0	0	0	1	x	Setzt Cursor an Position 0.
3	0	0	0	0	0	0	0	1	R/L	D/C	Eingabemodus: D0 wählt zwischen Verschieben der Anzeige (1) oder des Cursors (0); D1 wählt Richtung (1 rechts, 0 links)
4	0	0	0	0	0	0	1	D	C	B	Schaltet Display (D2), Cursor (D1) und Blinkfunktion (D0) ein bzw. aus
5	0	0	0	0	0	1	D/C	R/L	x	x	Verschiebt Display (D3=1) oder Cursor (D3=0) eine Stelle nach rechts (D2=1) oder links (D2=0).
6	0	0	0	0	1	4/8	Z	F	I	x	Setzt div. Einstellungen, s. Haupttext
7	0	0	0	1	A5	A4	A3	A2	A1	A0	Setzt Schreibadresse im CG-RAM
8	0	0	1	A6	A5	A4	A3	A2	A1	A0	Setzt Schreibadresse im DD-RAM
9	0	1	BF	A6	A5	A4	A3	A2	A1	A0	Busy-Flag (BF) und RAM-Adresszeiger (A6-A0) auslesen
10	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Schreibt Daten
11	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Liest Daten

Tabelle 4: Befehlsübersicht für die Ansteuerung von Displays auf Basis von HD44780-Controllern. Mit x markierte Bits werden ignoriert. Die Nummerierung der Befehle dient nur der Übersicht im Haupttext.

verhalten soll, wird durch Befehl 5 gesteuert. Damit die Anzeige verschoben wird, muss D3 auf 1 gesetzt werden, eine 0 an dieser Stelle sorgt dafür, dass der Cursor verschoben wird. Die Richtung der Verschiebung wird mit D2 festgelegt, eine 1 entspricht dabei einer Verschiebung nach rechts.

Zusätzlich zu dem automatischen Verschieben kann mit Befehl 3 der Cursor oder die Anzeige auch unabhängig von Schreibvorgängen nach rechts oder links verschoben werden.

Soll ein Zeichen angezeigt werden, welches nicht im vordefinierten Zeichensatz enthalten ist, können im CG-RAM (Character Generator) bis zu 8 eigene Zeichen definiert werden. Pro Zeile des 7×5 (bzw. 8×5) Zeichens geben die 5 unteren Bits eines Bytes im CG-RAM die Pixel an, die angezeigt werden sollen. Dabei steht das erste eigene Zeichen an den CG-RAM-Adressen 0 bis 7, das zweite bei 8 bis 15, usw.

5.4 Anpassungen an den Einsatz im Praktikum

Weil die Praktikumsteilnehmer nicht immer Erfahrung im Umgang mit Laborgeräten haben, sollte der EPROM-Programmer möglichst robust ausgeführt werden und Beschädigungen durch falsche Bedienung sollten vermieden werden.

Der Stromversorgungseingang des Geräts wurde gegen Verpolung, Überspannung und Anschluss an Wechselspannungsquellen abgesichert. Dazu dient eine Zener-Diode in Sperrichtung über den Anschlüssen für die Betriebsspannung, die bei Überspannung oder Verpolung leitend wird und eine Sicherung in der Stromversorgung durchbrennen lässt. Sowohl Diode als auch Sicherung können im Musteraufbau ohne Lötarbeiten ausgewechselt werden.

Die Bedienung des Geräts ist so einfach wie möglich gestaltet: Es gibt keine mehrfach belegten Tasten, die Bedienung ist „modeless“ (es gibt also keine Umschalt-, Meta- oder Funktionstasten) und die Rückmeldungen des Menüsystems sind eindeutig. Die aktuelle Funktion der vier Softkeys am oberen Rand der Tastatur wird im Display im Klartext angezeigt.

6 Programmierung und Arbeitsweise

6.1 Schreiben, Lesen und Löschen der PROMs

Um Daten in ein 2716-EPROM zu schreiben, muss das gewünschte 8 Bit lange Datenwort an die Ausgänge des Chips angelegt werden, während diese über eine 1 an $\overline{\mathrm{G}}$ hochohmig gehalten werden. Gleichzeitig wird an die Adresspins des Chips die gewünschte Adresse als 11 Bit langes Datenwort angelegt. An den V_{pp}-Anschluss wird die Brennspannung von 25 V angelegt, und durch einen kurzen[8] High-Puls an $\overline{\mathrm{E}}$P/CE werden die Daten geschrieben.

Die Programmiersequenz eines 2816-EEPROMs weicht davon etwas ab: An V_{pp} liegt hier grundsätzlich ein TTL-High-Pegel, also ca. 5 V. Nach dem Anlegen des Daten-

[8] Das Datenblatt schreibt eine Zeit von 50 ms vor

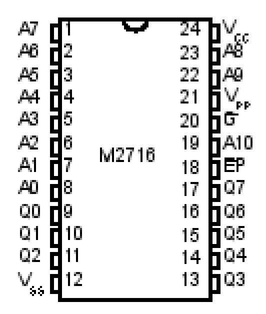


Abbildung 6: Pinbelegung von EPROMs des Typs 2716 (nach [Tho94]). Die mit A0 bis A10 bezeichneten Anschlüsse sind Adresspins, Q0 bis Q7 Datenpins.

und Adressworts muss V_{pp} für $1\,\mu$s auf Low-Pegel gebracht werden, um ein Datum zu schreiben.

Um die gespeicherten Inhalte auszulesen, muss die gewünschte Adresse wiederum als 11 Bit langes Datenwort an den Adressport des Chips gelegt werden. Liegt dann ein Low-Pegel an $\overline{\mathrm{G}}$ (Output Enable) und $\overline{\mathrm{E}}$P (Chip Enable) an, kann das ausgewählte Datum am Ausgang ausgelesen werden.

Zwischen dem Betrieb mit EPROMs und EEPROMs wird über den Schalter S1 umgeschaltet. Dieser Schalter wird vom Hauptcontroller abgefragt, indem der Pegel an Pin 17 überprüft wird. In der Schalterstellung „EPROM“ liegt dort ein High-Pegel an, sonst liegt der Pin an GND. Gleichzeitig legt der Schalter Pin 8 der IC-Fassung an die EPROM-Brennspannung von 25 V bzw. verbindet ihn zum Programmieren von EEPROMs mit Pin 21 des Hauptcontrollers.

6.2 Das Programm des Hauptcontrollers im Detail

Auf dem Hauptcontroller, also dem zentralen ATmega644, laufen alle Routinen zur Steuerung des Displays, des Brennvorgangs und der Kommunikation über RS232 bzw. TWI. Um diese relativ umfangreichen Funktionen einfach und leicht wartbar programmieren zu können, wurde das Programm des Hauptcontrollers in C geschrieben. Es umfasst kommentiert ca. 2000 Zeilen in 9 Dateien. Als Entwicklungsumgebung diente AVRStudio[9] mit WinAVR[10]. Später wurde unter Linux der AVR-Compiler gcc-avr mit der Bibliothek avr-libc und einer angepassten Makefile von WinAVR genutzt. Das kompilierte Programm wurde mittels des Tools avrdude[11] und eines ISP-Programmiergeräts mit USB-Anschluss auf den Controller geschrieben, der damit beim Programmieren in der Schaltung verbleiben konnte.

[9] http://atmel.com/dyn/products/tools_card.asp?tool_id=2725
[10] http://winavr.sourceforge.net/
[11] http://www.nongnu.org/avrdude, das entsprechende grafische Frontend avrdude-gui ist unter der Adresse http://sourceforge.net/projects/avrdude-gui/ verfügbar

Als Konvention wurde festgelegt, dass die Namen von Funktionen mit dem Namen ihrer Quelldatei beginnen. Die Funktion `error` aus der Datei `misc.c` heißt also `misc_error`.

6.2.1 Hauptprogramm und allgemeine Hilfsfunktionen

Legt man eine Versorgungsspannung an den Controller, startet er das geladene Programm automatisch. Wie bei C üblich, ist der Einsprungpunkt ins Programm die `main()`-Funktion. Sie liegt in der Datei `hauptcontroller.c` (A.1.2, s. S. 33). Dort werden einige hardwarespezifische Konstanten gesetzt, die das Ansprechen der Ports im Programm vereinfachen, und schließlich die Funktion `menu_main()` aufgerufen, die das Menü im Display anzeigt und die Abfrage von Tastendrücken veranlasst.

Die I/O-Pins der Atmel-Mikrocontroller werden in vier Ports, die je 8 Bit breit sind, unterteilt. Diese Ports können mit Hilfe der sogenannten DDR-Register (data direction register, für Port A etwa DDRA) zwischen Ein- und Ausgabe umgeschaltet werden. Dazu wird dem entsprechenden DDR-Register für jeden Pin des Ports, der als Ausgabe konfiguriert werden soll, ein 1-Bit zugewiesen. Die Umschaltung der Datenflussrichtung ist mit einigen Hilfsfunktionen in `misc.c` zusammengefasst.

6.2.2 Ansteuerung des Displays

Nach der Initialisierung von USART, TWI und dem Setzen einiger Kommunikationsparameter wird beim Start des Hauptcontrollers aus der Methode `init()` heraus auch das LC-Display initialisiert. Dazu wird die Funktion `lcd_init()` aufgerufen, die die Initialisierung und Parametrisierung des Displays übernimmt (s. S. 39).

Während die Dokumentation des Controllers vorschreibt, den Controller durch dreimaliges Senden von 0x30 (mit kurzen Pausen) zu initialisieren, hat sich im Praxistest gezeigt, dass dies nicht notwendig ist. Die Initialisierung wird daher im vorliegenden Quelltext durch einmaliges Senden von 0x30, gefolgt von 0x38 zur Auswahl des 8-Bit-Modus, erledigt. Dabei wird jeder der beiden Controller, die das Display steuern, getrennt initialisiert.

Nach erfolgter Initialisierung kann das Display zur Anzeige von Daten benutzt werden.

Beim Schreiben von Daten auf das Display muss programmintern bestimmt werden, welcher der beiden Controller im Display den Vorgang abwickeln soll. Die Entscheidung darüber kann einfach auf Basis der Zieladresse getroffen werden: Der Controller der oberen Displayhälfte verwaltet die Zeichen 0 bis 54, der der unteren Displayhälfte die übrigen. Diese Zuordnung wird in der Funktion `lcd_set_position` getroffen.

Listing 1: Auszug aus der `lcd.c`: Auswahl des für einen Schreibzugriff zuständigen Controllers

```
1  /*
2   * Setzt die Adresse (und damit die Position des Cursors) auf
3   * die uebergebene Position, um anschliessend dort schreiben zu
4   * koennen. Die Position muss zwischen 0 und 107 (4 * 27 - 1)
5   * liegen und wird umgerechnet auf die richtige Position im
6   * oberen bzw. unteren Controller.
7   */
```

```
 8 void lcd_set_position(uint8_t pos) {
 9     uint8_t ctl = 0;
10 
11     if (pos < 27) { // Position ist in Zeile 1
12 
13         // Der richtige Controller ist der obere, Umrechnung
14         // der Position nicht noetig.
15         ctl = LCD_CTL_TOP;
16 
17     } else if (pos < 54) { // Position ist in Zeile 2
18 
19         // Der richtige Controller ist der obere, Position muss
20         // umgerechnet werden (pos - 27 entspricht der Position
21         // in der Zeile, 64 ist der Offset fuer die 2. Zeile).
22         ctl = LCD_CTL_TOP;
23         pos = pos - 27 + 64;
```

Die in dieser Funktion bestimmten Konstanten werden dann an die Funktion `lcd_write` übergeben, die die Ansteuerung der Hardware übernimmt.

Listing 2: Ausschnitt aus der `lcd.h`: Umschaltung zwischen den beiden Controllern

```
1     PORTA = value;
2 
3     switch (controller) {
4 
5         case LCD_CTL_TOP:
6 
7             PORTC |= (1<<E1);
8             _delay_us(10);
9             PORTC &= ~(1<<E1);
```

In der `lcd.h` wurde die Konstante E1 so definiert, dass sie auf Pin 4 von Port C zeigt (E2 analog auf Pin 5). Nachdem ein Datenwort an Port A angelegt wurde, wird der gewünschte Displaycontroller durch Anlegen eines High-Pegels an Pin 4 (oberer Controller) oder Pin 5 (unterer Controller) von Port C ausgewählt. Das Programm wartet daraufhin 10 Mikrosekunden, um dem Controller genügend Zeit für die Übernahme der Daten zu geben, und deaktiviert den Controller dann wieder.

Nach dem Senden eines Befehls an den Controller muss zunächst geprüft werden, ob der Controller die Verarbeitung des letzten Befehls abgeschlossen hat und wieder bereit ist, bevor der nächste Befehl gesendet werden kann. Dazu wird aus dem Controller mittels RS=0 und R/W=1 das Busy-Flag (das 8. Bit des Antwort-Bytes) und die aktuelle Adresse im DD/CG-RAM gelesen. Ist der Controller beschäftigt, so ist das Busy-Flag auf 1 gesetzt, ansonsten kann der nächste Befehl gesendet werden. Diese Überprüfung wird von der Funktion `lcd_check_busy()` vorgenommen und muss für jeden Controller einzeln vorgenommen werden.

6.2.3 Zugriff auf die Speicherbausteine

Alle Funktionen für Zugriffe auf das PROM sind in der Quelldatei `eprom.c` zusammengefasst. Diese Datei stellt Funktionen zum Auslesen, Beschreiben und Leeren des Speicherbausteins bereit, ebenso auch Funktionen zur Abfrage, ob das PROM leer ist und ob die Schaltung im EPROM- oder EEPROM-Betrieb arbeitet.

Das Beschreiben der PROMs folgt dabei sowohl bei EPROMs als auch bei EEPROMs dem gleichen Schema: Das zu schreibende Datenwort wird vom Hauptcontroller an den (mit dem Display gemeinsam genutzten) Datenbus angelegt, die Zieladresse auf den Adressbus gelegt und daraufhin der Schreibbefehl an das PROM gegeben. Bei EPROMs wird der Schreibimpuls an den $\overline{\text{E}}$P/CE-Pin gelegt, bei EEPROMs an V_{pp}.

An einigen Stellen ist es notwendig, bestimmte Verzögerungen, etwa vor dem Anlegen der nächsten Adresse oder vor dem Absenden des nächsten Schreibimpulses, einzuhalten. Diese zumeist experimentell bestimmten Verzögerungszeiten sind ebenfalls in die `eprom.c` eingeflossen.

Listing 3: Die Brennroutine aus der `eprom.c`

```
1  /*
2   * Brennt den uebergebenen Wert an die uebergebene Adresse,
3   * eprom_write_start() muss vor dem ersten Aufruf aufgerufen
4   * werden.
5   */
6  void eprom_write(uint16_t address, uint8_t data) {
7
8      // Der Controller soll auf den LCD-/EPROM-Bus schreiben.
9      misc_set_data_direction(MISC_DD_UC);
10
11     // Die Adresse wird auf die Adress- und der Wert auf die
12     // Datenleitungen gelegt.
13     PORTB = address & 0xff;
14     PORTD &= ~(0x70);
15     PORTD |= (address>>4) & 0x70;
16     PORTA = data;
17
18     // Stabilisierung abwarten.
19     _delay_us(10);
20
21     switch (eprom_type) {
22
23         case EPROM_NEPROM: //EPROM
24
25             // Brennimpuls fuer 50ms anlegen.
26             PORTC |= (1<<EPCE);
27             _delay_ms(50);
28             PORTC &= ~(1<<EPCE);
29             break;
30
31         case EPROM_EEPROM: // EEPROM
```

```
32
33              // Brennimpuls geben und Stabilisierung abwarten.
34              PORTD &= ~(1<<VPPWE);
35              _delay_us(10);
36              PORTD |= (1<<VPPWE);
37              break;
38
39          // eprom_write() wurde aufgerufen, ohne dass
40          // vorher eprom_write_start() aufgerufen wurde.
41          default:
42              misc_error();
43              break;
44      }
45  }
```

6.2.4 TWI-Bus

Für die Kommunikation der beiden Controller untereinander wird das Two-Wire Serial Interface (TWI) benutzt. Die Geräte, die über diesen Bus kommunizieren, sind in einer logischen Master-Slave-Topologie angeordnet; der Bus ist multi-Master-fähig. Die Datenübertragung zwischen ihnen kann in beliebiger Richtung erfolgen, sowohl Master- als auch Slavegeräte können jeweils als Sender und Empfänger arbeiten. Das Mastergerät ist dabei immer für die Initialisierung der Kommunikation und die Erzeugung des Taktsignals zuständig.

Der Bus ist zwei Leitungen breit: SCL übermittelt das Taktsignal, während SDA für die Daten zuständig ist. Beide liegen über einen Pull-Up-Widerstand an V_{cc}, um definierte Pegelverhältnisse auf dem Bus zu garantieren. Die angeschlossenen Geräte erzeugen auf dem Bus Signale, indem sie Leitungen von diesem Pegel auf Masse ziehen. Dabei wird SDA in den entsprechenden Zustand (logisch 1 oder 0) gebracht, während SCL an Masse liegt. Für jedes Bit wird dann SCL einen Moment lang (abhängig von der Taktrate) an VCC gelegt, der Slave liest dabei den Wert von SDA aus.

Der TWI-Bus implementiert auch eine einfache Flusskontrollmöglichkeit für die angeschlossenen Geräte. Ein Slave, der mit der Verarbeitung der eingehenden Daten nicht schnell genug nachkommt, kann ein sendendes Gerät kurzzeitig unterbrechen, indem er die SCL-Leitung auf Masse zieht. Dadurch wird die Weitergabe des Taktsignals auf dem Bus unterbunden. Diese Flusskontrolle greift regelmäßig, wenn ein Slave-Gerät seine Adresse in einem Adresspaket (s.u.) erkennt. In diesem Fall muss der Slave zum Zeichen, dass er die Adresse erkannt hat und zur Kommunikation bereit ist, ein *acknowledgement* (kurz ACK) senden. Er blockiert dazu zunächst das Taktsignal, indem er SCL auf Massepotential zieht, bringt dann SDA auf Masse und erlaubt anschließend wieder die Taktausbreitung. Diese Signalabfolge stellt sicher, dass das ACK rechtzeitig beim Mastergerät ankommt: Würde der Slave die Kommunikation nicht unterbrechen, könnte es passieren, dass der Master die Kommunikation noch vor dem Erhalt des ACK durch einen timeout abbricht.

In der vorgestellten Schaltung wird nur die Master-Receiver-Konfiguration genutzt.

Dabei fragt der Hauptcontroller in regelmäßigen Abständen den Tastaturcontroller nach einer gedrückten Taste ab. Er initiiert dazu die Kommunikation mit einer sog. Start-Condition: als Mastergerät zieht der Hauptcontroller SDA auf Massepotential, während die Taktleitung SCL an V_{CC} liegt. Nach Senden der Start-Condition überträgt der Master ein Adresspaket, das aus 7 Adress-Bits und einem Read/Write-Bit besteht. Dieses Paket bestimmt, mit welchem anderen Gerät am Bus kommuniziert werden soll, und, über das Read/Write-Bit, ob dieses als Sender oder Empfänger auftreten soll.

Der Slave, zu dem die übertragene Adresse gehört, überträgt mit dem 9. Bit sein ACK, er legt dabei SDA an Masse, wenn er bereit für eine Übertragung ist und durch eine logische 1 im Read-/Write-Bit als Sender konfiguriert wurde. Hat der Slave sein ACK gesendet, liest der Master die nächsten 8 Datenbits von SDA, welches nun der Slave kontrolliert. Die Taktgenerierung wird weiterhin vom Master übernommen. Mit dem 9. Bit kann nun der Master ein ACK senden und so seine Bereitschaft zum Empfang eines weiteren Bytes signalisieren. Im vorliegenden Fall müssen aber keine weiteren Daten übertragen werden. Nach dem Empfang eines Bytes, mit dem eine eventuell gedrückte Taste übermittelt wird, sendet der Master kein ACK, lässt also SDA an V_{CC}, und beendet schließlich die Kommunikation mit einer sog. Stop-Condition. Dabei wird SDA von Masse auf V_{CC} gezogen, während gleichzeitig SCL ebenfalls an V_{CC} liegt. Anschließend kann eine neue Kommunikation gestartet werden.

6.2.5 Kommunikation mit dem PC

Trotz der angestrebten Unabhängigkeit von einem PC wurde eine Option entwickelt, die es erlaubt, das Programmiergerät vom PC aus zu bedienen. Dabei werden jedoch keine neuen Funktionalitäten hinzugefügt, sie dient lediglich als bequemere Alternative zu der eingebauten Tastatur sowie dem Display.

Das im Folgenden besprochene Kommunikationsprotokoll wurde im Hauptcontroller implementiert, es wurde jedoch kein Client für den PC entwickelt. Bei der Implementierung eines solchen Programms sollten die Hinweise in der Protokollbeschreibung beachtet werden, um eine sichere Kommunikation zu ermöglichen.

Die Kommunikation mit dem PC geschieht über eine serielle RS232-Schnittstelle. Als Kommunikationsparameter werden 8N1 bei einer Baudrate von 9600 genutzt, also 8 Bit Daten, kein Paritätsbit und zuletzt ein Stop-Bit zur Synchronisation auf das Übertragungsende. Das Protokoll wurde bewusst einfach gehalten und ausschließlich für die Kommunikation zwischen PC und Programmiergerät ausgelegt, so dass es sehr übersichtlich ist und nur kleine Datenmengen übertragen werden müssen. Das erleichtert die Verarbeitung im Mikrocontroller.

Um zu verhindern, dass der PC schneller sendet, als der doch vergleichsweise langsame Controller die Daten verarbeiten kann, wurde eine einfache Flusskontrolle im Protokoll verankert. Das Gerät sendet nach dem Empfang eines Bytes immer mindestens ein Byte zurück, worauf der PC Rücksicht nehmen muss. Weitere Befehle dürfen erst nach dieser Bestätigung abgesetzt werden.

Die Kommunikation beginnt, wenn die PC-Funktion am Programmiergerät über den entsprechenden Eintrag des Hauptmenüs aktiviert wird. Nun kann der PC das Signal

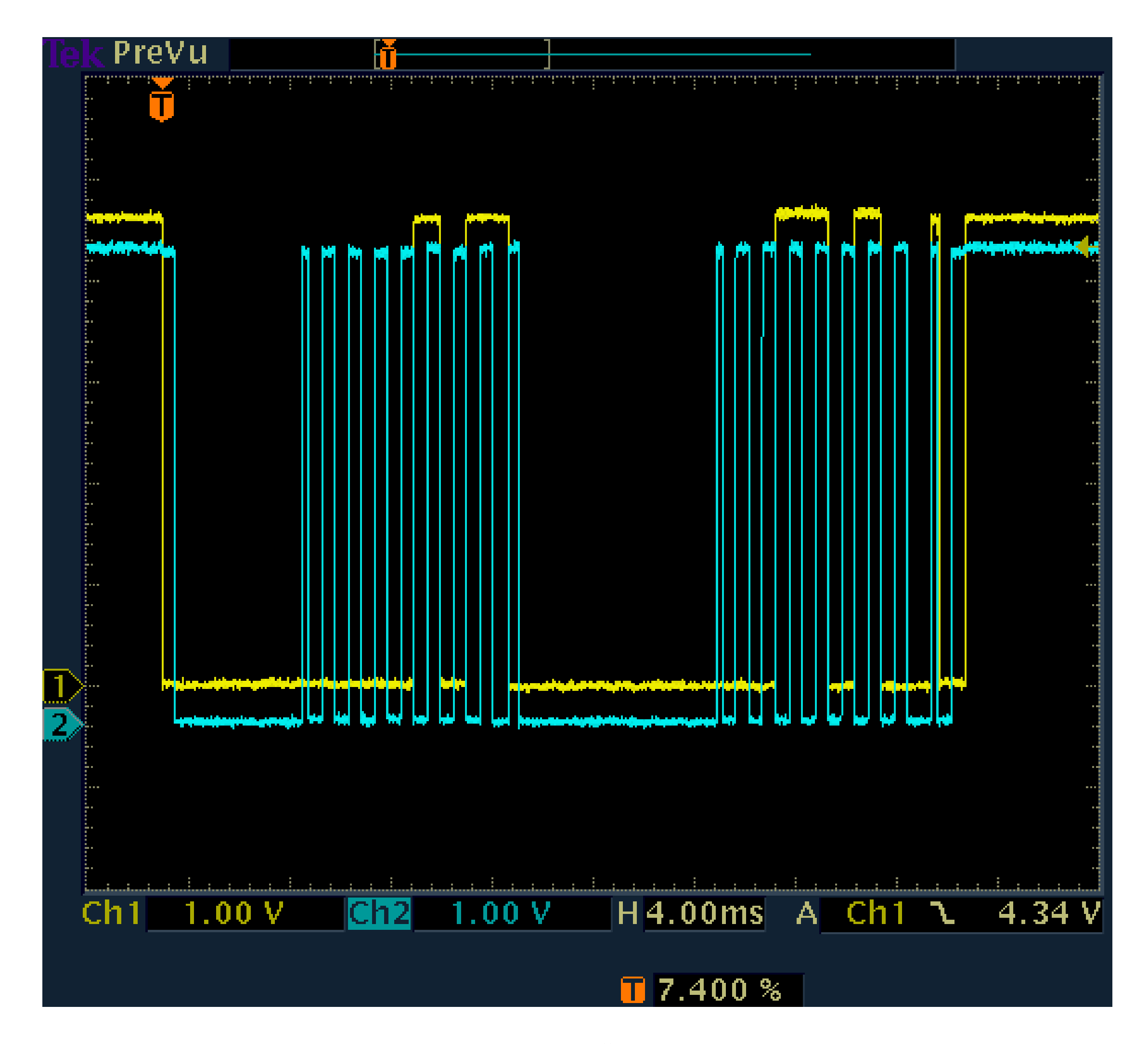


Abbildung 7: Dieses Oszillogramm zeigt die Übertragung eines Tastendrucks auf dem TWI-Bus. Das SDA-Signal ist auf Kanal 1 (gelb), das SCL-Signal auf Kanal 2 (türkis) aufgetragen. Um die Signale deutlicher erkennen zu können, sind ihre Nullpegel leicht gegeneinander verschoben. Links im Bild erkennt man, wie der Hauptcontroller SDA auf Massepegel zieht und damit die Übertragung einleitet. Es schließt sich der Austausch der Adressen und die Übertragung des Datenworts 00011010 (der Code der Taste BildAuf) an. Danach wird die Kommunikation per Stop-Condition beendet.

PC_CONNECT[12] senden. Daraufhin antwortet das Gerät mit PC_VERSION, also der Version des PC-Protokolls, die das Gerät benutzt. Der PC sollte die Protokollversion auswerten und, falls er die Version des Gerätes nicht unterstützt, eine entsprechende

[12] Alle Konstanten, die für das PC-Protokoll gebraucht werden, sind in der Datei pc.h des Hauptcontrollers zu finden, vgl. Listing A.1.17, S.76.

Fehlermeldung ausgeben und die Verbindung trennen. Dies geschieht durch Senden von PC_DISCONNECT und wird mit dem gleichen Wert vom Gerät bestätigt. Die Funktion zur Verbindungstrennung steht allerdings nur zur Verfügung, wenn das Gerät gerade keine andere Eingabe erwartet, ein begonnener (Nutz-)Datentransfer muss also vorher abgeschlossen werden.

Ist die Verbindung aktiv, erwartet das Gerät den nächsten Befehl. Nun können die verschiedenen Funktionen in beliebiger Reihenfolge genutzt werden, anschließend kehrt das Gerät wieder in den Wartezustand zurück und der nächste Befehl kann gegeben werden. Es sind die folgenden Befehle definiert:

- PC_DISCONNECT. Beendet die Kommunikation zwischen PC und Gerät. Das Gerät antwortet mit PC_DISCONNECT.

- PC_SIZE. Dieser Befehl fragt die Größe des Editorpuffers ab, also die nutzbare Größe des PROMs. Das Gerät antwortet mit 2 Byte, die die Größe repräsentieren. Dabei ist das erste Byte das *most significant byte* (MSB) der Puffergröße, das zweite das *least significant byte* (LSB). Bei Nutzung des ATmega644 ist dieser Wert typischerweise 2048, was auch der Größe des PROMs mit 2KiB entspricht.

- PC_INFO. Das Gerät antwortet mit Informationen über das eingesteckte PROM. Das erste Byte ist PC_NEPROM für ein 'normales' EPROM bzw. PC_EEPROM für ein EEPROM. Das zweite Byte teilt mit, ob das PROM leer (PC_EMPTY) oder nicht leer (PC_NEMPTY) ist. Da diese Funktion zur Abfrage des PROM-Typs die Methode `eprom_get_type()` (s. S. 47) verwendet, wird nicht erkannt, ob tatsächlich ein PROM eingesteckt ist, sondern genau genommen nur die Stellung des Wahlschalters S1 abgefragt.

- PC_CLEAR. Das Gerät leert das eingesteckte EEPROM, falls es nicht leer ist, indem es das EEPROM an jedem Byte mit 0xFF beschreibt, und bestätigt mit PC_CLEARED. Ist kein EEPROM eingesteckt oder ist es bereits leer, sendet es PC_NCLEARED.

- PC_FLUSH. Mit diesem Befehl wird der Editorpuffer geleert und das Gerät antwortet mit PC_FLUSHED.

- PC_READ. Das Gerät liest das eingesteckte PROM in den Editorpuffer aus und sendet daraufhin mit 2 Byte (MSB zuerst) die Anzahl der zu übertragenden Bytes. Anschließend überträgt es die ausgelesenen Daten des PROMs. Es beginnt dabei bei Adresse 0x00 und sendet jedes Byte, bis die zuvor übertragene Anzahl erreicht wurde. Alle nicht übertragenen Bytes bis zum Ende des Editorpuffers haben den Wert 0xFF, es werden also alle Bytes bis zum letzten von 0xFF verschiedenen Byte gesendet.

- PC_WRITE. Startet die Eingabe von Daten vom PC in den Editorpuffer. Das Gerät antwortet mit PC_WRITE_SIZE, der Aufforderung an den PC die zu übertragende Bytezahl zu senden. Nach dem Senden des MSB der Anzahl bestätigt

das Gerät den Empfang mit einem weiteren PC_WRITE_SIZE, woraufhin der PC das LSB sendet. Nun sendet das Gerät PC_WRITE_NEXT und der PC kann mit der Datenübertragung beginnen. Diese beginnt immer bei der Adresse 0x00 und endet nach dem Senden von der angekündigten Anzahl von Bytes. Nach jedem Byte bestätigt das Gerät den Empfang mit PC_WRITE_NEXT, nur nach dem letzten Byte antwortet es mit PC_WRITE_WRITTEN und ist nun wieder bereit für den nächsten Befehl.

- PC_BURN. Das eingesetzte PROM wird mit dem Inhalt des Editorpuffers beschrieben. Zu Beginn des Brennvorgangs wird PC_BURNING gesendet, zum Abschluss PC_BURNED.

Wird die PC-Funktion während einer Verbindung ausgeschaltet, sendet das Gerät PC_DISCONNECT, um den Verbindungsabbruch mitzuteilen.

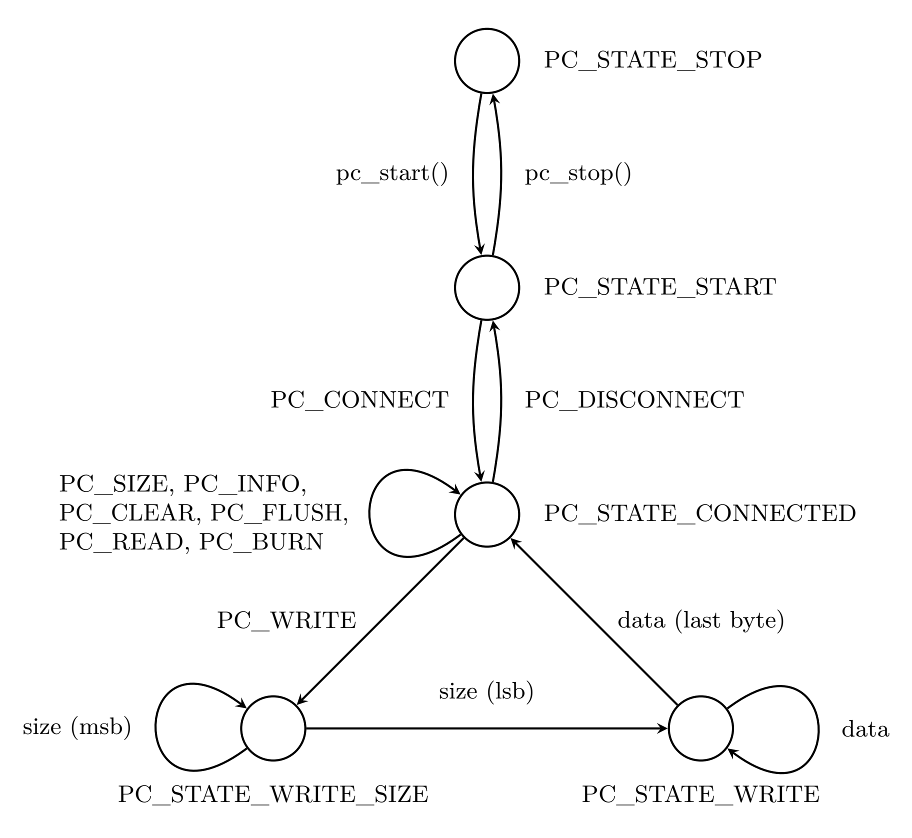


Abbildung 8: Protokollgraph der PC-Kommunikation.

6.2.6 Benutzerschnittstelle

Benutzer steuern das Programmiergerät über ein Menüsystem, das anhand von Tasteneingaben zwischen verschiedenen Ansichten wechselt. Das Menüsystem wertet die

Tastendrücke dabei selbst aus, gibt die Kontrolle aber an den Editor ab, wenn diese Funktion ausgewählt wird.

Der Editor ist aufgrund seiner Komplexität in eine eigene Quelldatei, die `editor.c`, ausgelagert worden (A.1.12, S. 61 f.). Das Ziel bei der Entwicklung war es, die Bedienung eng an Hexeditoren für den PC anzulehnen, damit die Nutzer möglichst nicht umlernen müssen.

Die Möglichkeiten des Editors sind durch die kleine Displaygröße etwas eingeschränkt: Da die unterste Zeile für die Beschriftung der Softkeys verwendet wird, bleiben nur drei Zeilen mit je 27 Zeichen für die Anzeige des eigentlichen Programms übrig. Jede Zeile enthält daher nur die Adresse des ersten angezeigten Halbbytes, ein Trennzeichen und danach die eingegebenen Halbbytes in Hexadezimalschreibweise. Auf Erläuterungen des Bildschirminhalts, etwa in Form einer Kopfzeile, wurde aus Platzgründen ebenso verzichtet wie auf Funktionen, für die ein größerer Ausschnitt des Programms sichtbar sein müsste, wie etwa Kopieren und Einfügen von Programmausschnitten.

Listing 4: In der Funktion `editor_display()` werden für jede der drei angezeigten Zeilen zuerst die Startadresse und danach die Daten in hexadezimaler Schreibweise ausgegeben.

```
for (j = 0; j < 3; j++) {

        // Initialisiere eine Zeile der Editoransicht und
        // trage Adresse (links) und Daten (rechts) ein
        char str[28] = "   h:                      ";
        str[0] = misc_halfbyte_to_char(
                (editor_start + j * 8)>>8);
        str[1] = misc_halfbyte_to_char(
                (editor_start + j * 8)>>4);
        str[2] = misc_halfbyte_to_char(
                editor_start + j * 8);

        uint8_t i = 0;

        for (i = 0; i < 8; i++) {
        str[editor_tabs[i]] = misc_halfbyte_to_char(
                editor_buffer[editor_start + j * 8 + i]>>4);
        str[editor_tabs[i] + 1] = misc_halfbyte_to_char(
                editor_buffer[editor_start + j * 8 + i]);
        }

        // Gibt den String im LCD aus.
        lcd_write_string(line[j], str);
}
```

6.3 Das Programm des Tastaturcontrollers im Detail

Der Tastaturcontroller wurde ebenfalls in C programmiert, wofür die gleiche Toolchain wie bei der Programmierung des Hauptcontrollers verwendet werden konnte. Sein Pro-

grammcode umfasst nur vier Dateien, die im Anhang wiedergegeben sind (A.2, S. 82).

6.3.1 Tastenabfrage

Die einzelnen Tasten der Tastatur sind in Form einer Matrix aus Zeilen- und Spaltenleitungen verdrahtet (vgl. 5.2, S. 11 f.). Der Tastaturcontroller legt nacheinander eine Spannung an die Gruppenleitungen an und prüft den Zustand der Spaltenleitungen. Wird die Taste am Kreuzungspunkt der Zeilen- und Spaltenleitung gedrückt, verbindet sie die beiden Leitungen, es wird also ein High-Pegel an der Spaltenleitung gemessen.

Für die Bedienung des Programmiergeräts ist es nie notwendig, mehrere Tasten gleichzeitig gedrückt zu halten. Da es nicht möglich ist, im Falle zweier gleichzeitig gedrückter Tasten zu entscheiden, welche vom Benutzer eigentlich gemeint war, wertet das Programm die Taste aus, die an der Gruppenleitung mit der niedrigeren Nummer liegt. Sollten beide Tasten an derselben Gruppenleitung liegen, wertet die Funktion `check_group` analog die Taste aus, die an der Zeilenleitung mit der niedrigsten Nummer liegt.

Listing 5: Prüfung auf Tastendrücke aus der `tastencontroller.c`.

```
1  uint8_t check_keystroke(void) {
2      // Nummer des Kabels der gedrueckten Taste
3      uint8_t wire;
4      // Nummer der aktuellen Tastengruppe
5      uint8_t group;
6  
7      // Alle Gruppen auf eine gedrueckte Taste ueberpruefen,
8      // vorzeitig abbrechen, sobald eine gefunden wurde
9      for (group = 0; group < 4; group++) {
10         wire = check_group(groups[group]);
11 
12         if (wire != 0) { // Eine Taste wurde gedrueckt
13             break;
14         }
15     }
16 
17     if (wire != 0) { // Eine Taste wurde gedrueckt
18         // Die zu der Gruppe und dem Kabel gehoerende Taste
19         // zurueckgeben
20         return keys[group * 8 + wire - 1];
21     } else { // Keine Taste wurde gedrueckt
22         return KEY_NONE;
23     }
24 
25 }
```

6.3.2 Kommunikation mit dem Hauptcontroller

Erkennt der Tastaturcontroller einen Tastendruck, wird der Code der gedrückten Taste über eine TWI-Verbindung an den Hauptcontroller gemeldet. Der ATtiny2313 unter-

stützt allerdings das auf dem ATmega644 bereits implementierte TWI nicht direkt. Statt dessen ist ein vereinfachtes Bussystem, das sogenannte USI (Universal Serial Interface), implementiert. Dieses befindet sich auf einer etwas niedrigeren, hardware-näheren Abstraktionsebene. Die übertragenden Daten werden vom Controller nicht vorverarbeitet, sondern müssen vom laufenden Programm interpretiert werden. Der Controller kann damit nur mit Hilfe zusätzlicher Software mit TWI-Geräten kommunizieren. Die Funktionen der `usi.c` implementieren das TWI-Protokoll auf dem USI und stellen die Schnittstelle für die Datenübertragung dar.

7 Bedienung

Nach dem Einschalten des Geräts wird in der untersten Displayzeile das Hauptmenü angezeigt. Die einzelnen Menüpunkte können mit den zugeordneten Softkeys direkt unter dem Display angewählt werden.

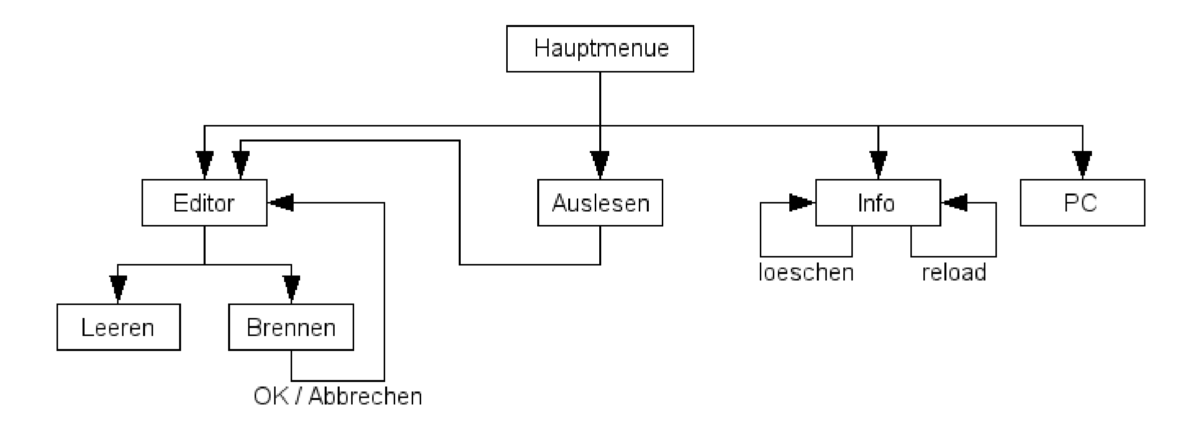


Abbildung 9: Das Menüsystem des Geräts. Von allen Untermenüs aus kann über den rechten Softkey (dann mit „zurück“ beschriftet) ins nächsthöhere Menü zurückgewechselt werden.

Aus dem Hauptmenü heraus kann über den Menüpunkt „Editor“ in eine Editoransicht gewechselt werden, daneben können Informationen über ein eingesetztes EPROM oder EEPROM abgerufen und das Gerät auf PC-Steuerung umgeschaltet werden. Zusätzlich gibt es eine Funktion „Auslesen“, bei der zuerst das eingesteckte PROM ausgelesen und anschließend in die Editoransicht gewechselt wird.

Der Editor dient zur Eingabe eines neuen Programms. Hier gemachte Eingaben bleiben erhalten, auch wenn in ein anderes Menü gewechselt wird. Um mit der Eingabe eines neuen Programms zu beginnen, kann mit dem Softkey „Leeren“ der komplette Inhalt des Editors gelöscht werden. Der Softkey „Brennen“ brennt nach einer Sicherheitsabfrage das geschriebene Programm ins eingelegte PROM, wobei abhängig von der Stellung des EPROM-/EEPROM-Umschalters auf der Hauptplatine automatisch der richtige Schreibmodus gewählt wird. Über den Softkey „zurueck“ gelangt man wieder ins Hauptmenü;

der Inhalt des Editors wird dabei, wie erwähnt, nicht gelöscht.

Der Auslesen-Modus ruft, wie erwähnt, den Editor auf, nachdem der Inhalt des PROMs in den Editorpuffer gelesen wurde. Er ist für die Kontrolle des gebrannten Programms, aber auch zum bequemen Kopieren gedacht: Legt man ein beschriebenes PROM ein und drückt auf „Auslesen", werden die im PROM abgelegten Daten in den Editor übernommen. Sie können dann in ein neues, noch leeres PROM gebrannt und ggf. vorher noch editiert werden.

In der Editoransicht lehnt sich die Bedienung an PC-basierte Texteditoren an. Die Pfeiltasten bewegen den Cursor, die Tasten Page Up und Page Down scrollen die Ansicht um eine Bildschirmhöhe.



Abbildung 10: Die Editoransicht. In der linken Spalte wird die Adresse des jeweils ersten Bytes der Zeile hexadezimal angezeigt, es folgen 8 Byte Daten. Der Cursor steht im Bild an der Adresse 008h. In der untersten Zeile sind die aktuellen Belegungen der Softkeys 1, 2 und 4 angegeben.

Der Menüpunkt „Info" prüft, ob das eingesetzte PROM leer (also nicht beschrieben) ist und gibt eine kurze Meldung aus. Dabei ist ein PROM leer, wenn kein Byte einen anderen Wert als 0xFF hat. Zusätzlich können hier nicht-leere EEPROMs gelöscht werden, also komplett mit 0xFF beschrieben werden.

Eine Bedienungsanleitung für den Einsatz im Praktikum ist im Anhang enthalten (Anhang B, S. 94).

8 Fehlerbehebung und Reparaturen

8.1 Fehler im Betrieb

Sollte das Gerät im Betrieb einmal in den Fehlerzustand geraten (die Fehlerkontroll-LED auf der Platine blinkt), kann ein Reset einfach durch kurzes Trennen von der Betriebsspannung ausgelöst werden. Ein solcher Reset empfiehlt sich auch bei Kommunikationsproblemen zwischen den beiden Controllern, wenn also beispielsweise keine Tastendrücke mehr akzeptiert werden.

8.2 Ersatzteilbeschaffung

Die verwendeten Mikrocontroller sind derzeit so weit verbreitet, dass auch nach dem Auslaufen der Serie pinkompatible Ersatztypen zur Verfügung stehen dürften. Auch die übrigen verwendeten Bauteile sind Standardprodukte, die noch einige Zeit im Elektronikhandel erhältlich sein dürften.

Das Display dagegen stammt von einem Restpostenhändler, so dass es schwierig werden könnte, einen elektrisch und mechanisch passsenden Ersatztyp zu beschaffen. Da das Display aber zwei der verbreiteten HD44780-Controller verwendet, können alle Displays mit demselben Grundkonzept ohne Änderungen am Programm verwendet werden.

Aber auch die Speicherbausteine der Typen 2716 und 2816 sind mittlerweile schwer zu beschaffen, weil die Entwicklung hin zu PROMs mit deutlich größerem Speicherplatz geht. Es gibt vereinzelt noch pinkompatible Bausteine der Typen U556 und K573RF2 sowie einen pinkompatiblen SRAM-Baustein mit der Bezeichnung U 6516[FA89] aus DDR-Produktion, der zumindest in den Applikationsschaltungen die Aufgaben des 2716 übernehmen kann.

9 Zusammenfassung und Ausblick

Das Programmiergerät ist in der beschriebenen Ausführung weitgehend an den vorgesehenen Einsatz im Hardwarepraktikum angepasst. Es lässt sich allerdings in gewissen Grenzen an andere Einsatzsituationen anpassen.

9.1 Aufbau ohne Gehäuse

Geeignete Pultgehäuse mit Platz für eine Tastatur und das Display sind realtiv teuer, Eigenbaugehäuse mit viel Arbeitsaufwand verbunden. Ein Aufbau der oben beschriebenen Platinen ohne Gehäuse ist aber schwierig, da die Tastaturcontroller-Platine und die Fassung des EPROM-Sockels relativ klein und unhandlich sind. Außerdem ist für die Tastatur keine Platine vorgesehen.

Auf einer ganzen Europlatine sollte, wenn man kleine Tasten mit kleinen Abständen in Kauf nimmt, die komplette Schaltung Platz finden. Da die verwendete Version des Platinenlayoutprogramms EAGLE nur die halbe Europlatinengröße unterstützt, wurde für diese Version kein Layout erstellt.

Alternativ könnte die Schaltung auch auf zwei halbe Europlatinen verteilt werden, die auf die verbreiteten Laborgehäuse montiert werden können und neben der Stromversorgung nur durch die zwei Drähte des I^2C-Busses verbunden sein müssten.

Durch den RS232-Anschluss kann der Brenner, wie die meisten kommerziell angebotenen Geräte, aber auch ganz ohne Tastatur und Display verwendet werden. Damit lässt sich das gesamte Gerät in ein kompaktes Gehäuse von etwa $9 \times 12 \times 3\,\mathrm{cm}^3$ einbauen, benötigt allerdings einen PC mit entsprechender Software für die Ansteuerung.

9.2 Erweiterungen

Die Anpassung an andere PROM-Typen ist schwieriger: Da nicht alle Pins des PROMs mit dem Controller verbunden sind, lässt sich die Anpassung an Typen, die nicht pinkompatibel zum 2716 sind, nur teilweise in Software vornehmen. Die Pins 12 (U_{ss}), 18 (PGM), 20 ($\overline{OE}$), 21 (U_{pp}) und 24 (U_{cc}) sind mit Hilfs- und Versorgungsspannungen belegt, so dass EPROMs, die auf einem dieser Pins eine Datenleitung haben, nur nach einer entsprechenden Überarbeitung der Platine verwendet werden können. Viele 16k-EPROMs sind aber pinkompatibel zum 2716, so dass bei ihnen nur auf die Brennspannung geachtet werden muss: Der Programmer stellt 25 V und 5 V bereit, zwischen den beiden Spannungen wird über den Schalter S1 umgeschaltet. Der Software wird die Schalterstellung über Port D3 mitgeteilt: Er liegt an V_{cc}, wenn die Ladungspumpe zugeschaltet ist, sonst an GND.

Prinzipiell wäre auch eine Kaskadierung mehrerer Programmer vorstellbar, so dass man ein Gerät für die Eingabe des Programms verwendet und dieses Gerät dann das Programm und die Schreibbefehle an andere, über RS232 angeschlossene Geräte weitergibt, die das Programm dann parallel in mehrere EPROMs schreiben könnten.

9.3 PC-Steuerung

Der Programmer ist in Hard- und Software für die Fernsteuerung durch ein PC-Programm vorbereitet, so dass er wie die weit verbreiteten Programmiergeräte ohne eigene Tastatur eingesetzt werden kann. Dazu ist auf PC-Seite ein Programm nötig, das das in der `pc.c` beschriebene Protokoll beherrscht und mit dem Controller über die serielle Schnittstelle kommuniziert.

Anhang

A Quelltexte

A.1 Hauptcontroller

A.1.1 hauptcontroller.h

```
#ifndef __HAUPTCONTROLLER_H__
#define __HAUPTCONTROLLER_H__

#include <avr/io.h>
#include "menu.h"
#include "usart.h"
#include "twi.h"
#include "lcd.h"
#include "editor.h"

/*
    Konstante, mit der die Baud-Rate fuer die
    USART-Uebertragung festgelegt wird.
*/
#define USART_BAUD 9600

/*
    Konstante, mit der festgelegt wird, dass die interne
    Verdoppelung der USART-Baud-Rate genutzt werden soll. Dies
    fuehrt zu einem niedrigeren Wert im UBRR fuer die gleiche
    Baud-Rate und einer deutlichen Fehler-Verringerung bei der
    Uebertragung (0,2% statt 7%).
*/
#define USART_DOUBLE 1

#endif
```

A.1.2 hauptcontroller.c

```
#include "hauptcontroller.h"

/*
 *   Initialisiert den Controller und die angeschlossenen
 *   Komponenten.
 */
void init(void) {

    // Aktiviert USART und TWI.
    usart_init(USART_BAUD, USART_DOUBLE);
    twi_init();
```

```
12
13      /*
14          Setzt die initialen Datenrichtungen der Pins am
15          Controller. Deaktiviert den Output am LCD-/EPROM-Bus
16          (Port A) und am EPROM/EEPROM-Schalter (Pin D3).
17          Aktiviert den Output fuer die EPROM-Adressleitungen
18          (Port B und Pins D4-D6), fuer die LCD- und
19          EPROM-Steuerleitungen (Pins C2-C7 und D7) und fuer
20          die LED (Pin D2).
21      */
22      DDRA = 0x00;
23      DDRB = 0xff;
24      DDRC |= 0xfc;
25      DDRD &= ~(1<<DDD3);
26      DDRD |= 0xf4;
27
28      /*
29          Setzt die initialen Daten bzw. Pull-Up-Widerstaende.
30          Deaktiviert die Pull-Up-Widerstaende am LCD-/EPROM-Bus
31          (Port A) und am EPROM/EEPROM-Schalter (Pin D3). Setzt
32          die EPROM-Adressleitungen (Port B und Pins D4-D6), die
33          LCD-Steuerleitungen (Pins C2-C5) und die Brennleitung
34          des EPROMs (Pin C7) auf 0. Setzt die Output-Leitung des
35          EPROMs (Pin C6), die Brennspannungsleitung (Pin D7) und
36          die LED-Leitung (Pin D2) auf 1.
37      */
38      PORTA = 0x00;
39      PORTB = 0x00;
40      PORTC &= 0x03;
41      PORTC |= 0x40;
42      PORTD &= 0x03;
43      PORTD |= 0x84;
44
45      // Initialisiert das LCD und den Editor.
46      lcd_init();
47      editor_init();
48  }
49
50  /*
51   * Wird beim Starten des Controllers aufgerufen. Initialisiert
52   * den Controller und uebergibt die Kontrolle an das
53   * Hauptmenue.
54   */
55  int main(void) {
56      init();
57      menu_main();
58      return 0;
59  }
```

A.1.3 misc.h

```
#ifndef __MISC_H__
#define __MISC_H__

#include <avr/io.h>
#include <inttypes.h>
#include <util/delay.h>

// Konstanten fuer die Datenrichtung am LCD-/EPROM-Bus
#define MISC_DD_OFF 0
#define MISC_DD_UC 1
#define MISC_DD_EPROM 2
#define MISC_DD_LCD 3

// Konstanten, die den Pins am Controller Namen geben, die ihre
// Funktion widerspiegeln.
#define GOE PORTC6
#define RW PORTC3
#define LED PORTD2
#define EPCE PORTC7
#define EPEEP PD7
#define VPPWE PD7

void misc_error(void);

void misc_set_data_direction(uint8_t dd);

uint8_t misc_halfbyte_to_char(uint8_t hb);

#endif
```

A.1.4 misc.c

```
#include "misc.h"

/*
 * Erzeugt eine Fehlermeldung, indem das Programm in eine
 * Endlosschleife gefuehrt wird, in der die LED immer wieder
 * an- und ausgeschaltet wird. Eine Rueckkehr aus diesem
 * Zustand ist nur durch einen Reset des Controllers moeglich.
 */
void misc_error(void) {

    // Erzeugen einer Endlosschleife, damit die normale
    // Programmausfuehrung verhindert wird.
    while (1) {

        // Schalte die LED ein und warte 0,5 Sekunden.
        PORTD &= ~(1<<LED);
```

```
17          _delay_ms(250);
18          _delay_ms(250);
19
20          // Schalte die LED aus und warte 0,5 Sekunden.
21          PORTD |= (1<<LED);
22          _delay_ms(250);
23          _delay_ms(250);
24      }
25
26  }
27
28  /*
29   * Aktiviert oder deaktiviert den Output des Controllers am
30   * LCD-/EPROM-Bus.
31   */
32  void misc_set_dd_uc(uint8_t enable) {
33
34      // Falls enable nicht 0 ist, soll der Output aktiviert
35      // werden.
36      if (enable) {
37          DDRA = 0xff;
38      } else { // Der Output soll deaktiviert werden
39          DDRA = 0;
40          PORTA = 0;
41      }
42
43  }
44
45  /*
46   * Aktiviert oder deaktiviert den Output des EPROMs am
47   * LCD-/EPROM-Bus.
48   */
49  void misc_set_dd_eprom(uint8_t enable) {
50
51      // Falls enable nicht 0 ist, soll der Output aktiviert
52      // werden.
53      if (enable) {
54          PORTC &= ~(1<<GOE);
55      } else { // Der Output soll deaktiviert werden
56          PORTC |= (1<<GOE);
57      }
58
59  }
60
61  /*
62   * Aktiviert oder deaktiviert den Output des LCDs am
63   * LCD-/EPROM-Bus.
64   */
65  void misc_set_dd_lcd(uint8_t enable) {
```

```
66
67      // Falls enable nicht 0 ist, soll der Output aktiviert
68      // werden.
69      if (enable) {
70          PORTC |= (1<<RW);
71      } else { // Der Output soll deaktiviert werden
72          PORTC &= ~(1<<RW);
73      }
74
75  }
76
77  // Speichert aktuellen Zustand des LCD-/EPROM-Busses.
78  uint8_t misc_data_direction = 0xff;
79
80  /*
81   * Setzt die Datenrichtungen der Pins des Controllers, des LCDs
82   * und des EPROMs am LCD-/EPROM-Bus, so dass nur einer der
83   * drei Teilnehmer darauf schreiben kann.
84  */
85  void misc_set_data_direction(uint8_t dd) {
86
87      // Ist die neue Datenrichtung bereits gesetzt, tue nichts.
88      if (misc_data_direction == dd) {
89          return;
90      }
91
92      switch (dd) {
93
94          // Niemand soll auf den Bus schreiben.
95          case MISC_DD_OFF:
96
97              // Deaktiviere alle Outputs auf dem Bus.
98              misc_set_dd_uc(0);
99              misc_set_dd_eprom(0);
100             misc_set_dd_lcd(0);
101             break;
102
103         // Der Controller soll auf den Bus schreiben.
104         case MISC_DD_UC:
105
106             misc_set_dd_eprom(0);
107             misc_set_dd_lcd(0);
108             misc_set_dd_uc(1);
109             break;
110
111         // Das EPROM soll auf den Bus schreiben.
112         case MISC_DD_EPROM:
113
114             misc_set_dd_uc(0);
```

```
115                 misc_set_dd_lcd(0);
116                 misc_set_dd_eprom(1);
117                 break;
118 
119             // Das LCD soll auf den Bus schreiben.
120             case MISC_DD_LCD:
121 
122                 misc_set_dd_uc(0);
123                 misc_set_dd_eprom(0);
124                 misc_set_dd_lcd(1);
125                 break;
126 
127             // Der Parameter dd hat einen falschen Wert.
128             default:
129                 misc_error();
130                 break;
131 
132         }
133 
134         // Speichere den neuen Zustand.
135         misc_data_direction = dd;
136     }
137 
138     /*
139      * Wandelt das uebergebene halbe Byte (die 4 hoechstwertigen
140      * Bits werden ignoriert) in das zugehoerige Hex-Zeichen im
141      * ASCII-Format um, welches dann am LCD angezeigt werden kann.
142      */
143     uint8_t misc_halfbyte_to_char(uint8_t hb) {
144 
145         // Ignoriere die 4 hoechstwertigen Bits,
146         // hb hat also einen Wert zwischen 0 und 16.
147         hb &= 0x0f;
148 
149         // Falls das halbe Byte kleiner als 10 ist, ist es eine
150         // Ziffer in der Hex-Darstellung.
151         if (hb < 10) {
152 
153             // Addiere den Offset 0x30, um aus der Zahl hb den
154             // ASCII-Code zu erhalten, der hb darstellt.
155             hb += 0x30;
156 
157         } else { // Das halbe Byte ist ein Buchstabe
158 
159             // Addiere den Offset 0x37, um den entsprechenden
160             // Buchstaben A-F zu erhalten.
161             hb += 0x37;
162         }
163 
```

```
    return hb;
}
```

A.1.5 lcd.h

```
#ifndef __LCD_H__
#define __LCD_H__

#include <avr/io.h>
#include <inttypes.h>
#include <util/delay.h>
#include "misc.h"

#define LCD_CTL_TOP 0
#define LCD_CTL_BOTTOM 1
#define LCD_IR 0
#define LCD_DR 1
#define LCD_BUSY 0x80
#define LCD_LINE1 0
#define LCD_LINE2 27
#define LCD_LINE3 54
#define LCD_LINE4 81

#define RS PORTC2
#define E1 PORTC4
#define E2 PORTC5

void lcd_init(void);

void lcd_write_char(uint8_t pos, uint8_t ch);

void lcd_write_string(uint8_t pos, char str[]);

void lcd_clear(void);

void lcd_set_position(uint8_t pos);

void lcd_display_cursor(uint8_t cursor);

#endif
```

A.1.6 lcd.c

```
#include "lcd.h"

// Speichert den aktuellen Status des Cursors. Ist Bit 0
// gesetzt, so wird der Cursor angezeigt, ansonsten nicht. Ist
//  Bit 1 gesetzt, so wird der Cursor im unteren Controller
//  angezeigt, sonst im oberen.
uint8_t lcd_cursor = 0;
```

```
8
9  /*
10  * Liest den Wert des uebergebenen Registers des uebergebenen
11  * Controllers aus und gibt ihn zurueck.
12  */
13 uint8_t lcd_read(uint8_t controller, uint8_t reg) {
14     uint8_t value = 0;
15
16     // Das LCD soll auf den LCD-/EPROM-Bus schreiben.
17     misc_set_data_direction(MISC_DD_LCD);
18
19     switch (reg) {
20
21         // Aus dem Befehlsregister soll gelesen werden.
22         case LCD_IR:
23             PORTC &= ~(1<<RS);
24             break;
25
26         // Aus dem Datenregister soll gelesen werden.
27         case LCD_DR:
28             PORTC |= (1<<RS);
29             break;
30
31         // Ein falscher Wert wurde uebergeben.
32         default:
33             misc_error();
34             break;
35
36     }
37
38     switch (controller) {
39
40         // Aus dem oberen Controller soll gelesen werden.
41         case LCD_CTL_TOP:
42
43             // Oberen Controller aktivieren, Stabilisierung
44             // abwarten, den gelesenen Wert speichern und den
45             // Controller wieder deaktivieren.
46             PORTC |= (1<<E1);
47             _delay_us(10);
48             value = PINA;
49             PORTC &= ~(1<<E1);
50             break;
51
52         // Aus dem unteren Controller soll gelesen werden.
53         case LCD_CTL_BOTTOM:
54
55             // Unteren Controller aktivieren, Stabilisierung
56             // abwarten, den gelesenen Wert speichern und den
```

```
57              // Controller wieder deaktivieren.
58              PORTC |= (1<<E2);
59              _delay_us(10);
60              value = PINA;
61              PORTC &= ~(1<<E2);
62              break;
63
64          // Ein falscher Wert wurde uebergeben.
65          default:
66              misc_error();
67              break;
68      }
69
70      return value;
71  }
72
73  /*
74   * Blockt, bis das Busy-Flag des uebergebenen Controllers nicht
75   * mehr gesetzt ist.
76   */
77  void lcd_check_busy(uint8_t controller) {
78
79      while (lcd_read(controller, LCD_IR) & LCD_BUSY);
80
81  }
82
83  /*
84   * Schreibt value in das uebergebene Register des
85   * uebergebenen Controllers.
86   */
87  void lcd_write(uint8_t controller, uint8_t reg, uint8_t value) {
88
89      // Der Controller soll auf den LCD-/EPROM-Bus schreiben.
90      misc_set_data_direction(MISC_DD_UC);
91
92      switch (reg) {
93
94          // Schreibziel: Befehlsregister
95          case LCD_IR:
96              PORTC &= ~(1<<RS);
97              break;
98
99          // Schreibziel: Datenregister
100         case LCD_DR:
101             PORTC |= (1<<RS);
102             break;
103
104         // Ein falscher Wert wurde uebergeben.
105         default:
```

```
106                 misc_error();
107                 break;
108 
109         }
110 
111         // Den Wert auf die Datenleitung legen.
112         PORTA = value;
113 
114         switch (controller) {
115 
116             // Es soll in den oberen Controller geschrieben werden
117             case LCD_CTL_TOP:
118 
119                 // Oberen Controller aktivieren, Stabilisierung
120                 // abwarten und den Controller wieder deaktivieren.
121                 PORTC |= (1<<E1);
122                 _delay_us(10);
123                 PORTC &= ~(1<<E1);
124                 break;
125 
126             // Es soll in den unteren Controller geschrieben werden
127             case LCD_CTL_BOTTOM:
128 
129                 // Unteren Controller aktivieren, Stabilisierung
130                 // abwarten und den Controller wieder deaktivieren.
131                 PORTC |= (1<<E2);
132                 _delay_us(10);
133                 PORTC &= ~(1<<E2);
134                 break;
135 
136             // Ein falscher Wert wurde uebergeben.
137             default:
138                 misc_error();
139                 break;
140 
141         }
142 
143         // Warten, bis der Controller fertig ist.
144         lcd_check_busy(controller);
145 
146     }
147 
148     /*
149      * Initialisiert das LCD mit keinem Text und keinem Cursor.
150      */
151     void lcd_init() {
152 
153         // Warten, bis das LCD nach dem Einschalten bereit ist.
154         _delay_ms(20);
```

```
155
156     // Beide Controller auf 8-Bit-Datenbreite setzen, dabei
157     // zuerst auf ein-zeilig, dann auf zwei-zeilig setzen.
158     lcd_write(LCD_CTL_TOP, LCD_IR, 0x30);
159     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x30);
160     lcd_write(LCD_CTL_TOP, LCD_IR, 0x38);
161     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x38);
162
163     // Die Anzeige leeren.
164     lcd_write(LCD_CTL_TOP, LCD_IR, 0x01);
165     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x01);
166
167     // Den Cursor an den Anfang setzen.
168     lcd_write(LCD_CTL_TOP, LCD_IR, 0x02);
169     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x02);
170
171     // Beim Schreiben eines Zeichens den Cursor nach rechts
172     // verschieben statt der ganzen Anzeige.
173     lcd_write(LCD_CTL_TOP, LCD_IR, 0x06);
174     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x06);
175
176     // Den Cursor abschalten.
177     lcd_write(LCD_CTL_TOP, LCD_IR, 0x0c);
178     lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x0c);
179 }
180
181 /*
182  * Setzt die Adresse (und damit die Position des Cursors) auf
183  * die uebergebene Position, um anschliessend dort schreiben zu
184  * koennen. Die Position muss zwischen 0 und 107 (4 * 27 - 1)
185  * liegen und wird umgerechnet auf die richtige Position im
186  * oberen bzw. unteren Controller.
187  */
188 void lcd_set_position(uint8_t pos) {
189     uint8_t ctl = 0;
190
191     if (pos < 27) { // Position ist in Zeile 1
192
193         // Der richtige Controller ist der obere, Umrechnung
194         // der Position nicht noetig.
195         ctl = LCD_CTL_TOP;
196
197     } else if (pos < 54) { // Position ist in Zeile 2
198
199         // Der richtige Controller ist der obere, Position muss
200         // umgerechnet werden (pos - 27 entspricht der Position
201         // in der Zeile, 64 ist der Offset fuer die 2. Zeile).
202         ctl = LCD_CTL_TOP;
203         pos = pos - 27 + 64;
```

```
204
205     } else if (pos < 81) { // Position ist in Zeile 3
206
207         // Der richtige Controller ist der untere, Position
208         // muss umgerechnet werden (pos - 54 entspricht der
209         // Position in der Zeile, kein Offset fuer die 1.
210         // Zeile).
211         ctl = LCD_CTL_BOTTOM;
212         pos = pos - 54;
213
214     } else if (pos < 108) { // Position ist in Zeile 4
215
216         // Der richtige Controller ist der untere, Position
217         // muss umgerechnet werden (pos - 81 entspricht der
218         // Position in der Zeile, 64 ist der Offset fuer die 2.
219         // Zeile).
220         ctl = LCD_CTL_BOTTOM;
221         pos = pos - 81 + 64;
222
223     } else { // keine gueltige Position
224         misc_error();
225     }
226
227     // Schreibe die umgerechnete Position in den richtigen
228     // Controller (Bit 7 gehoert nicht zur eigentlichen
229     // Position, sondern ist der Befehl zum Schreiben der
230     // Position und muss 1 sein).
231     lcd_write(ctl, LCD_IR, pos | 0x80);
232
233     // Falls der Cursor angezeigt wird, muss geprueft werden,
234     // ob der Cursor aufgrund der neuen Position den Controller
235     // wechseln muss.
236     if (lcd_cursor & 0x01) {
237
238         if (lcd_cursor & 0x02 && ctl == LCD_CTL_TOP) {
239
240             // Der Cursor wird im unteren Controller angezeigt,
241             // die neue Position befindet sich aber im oberen.
242             // Dann: Speichere den Status, deaktiviere den
243             // Cursor im  unteren und aktiviere ihn im oberen
244             // Controller.
245             lcd_cursor = 0x01;
246             lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x0c);
247             lcd_write(LCD_CTL_TOP, LCD_IR, 0x0e);
248
249         } else if (!(lcd_cursor & 0x02) &&
250                 ctl == LCD_CTL_BOTTOM) {
251             // Der Cursor wird im oberen Controller angezeigt,
252             // die neue Position befindet sich aber im unteren.
```

```
253                 // Dann: Speichere den Status, deaktiviere den
254                 // Cursor im oberen und aktiviere ihn im unteren
255                 // Controller.
256                 lcd_cursor = 0x03;
257                 lcd_write(LCD_CTL_TOP, LCD_IR, 0x0c);
258                 lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x0e);
259
260             }
261
262         }
263
264     }
265
266     /*
267      * Schreibt das uebergebene Zeichen an die angegebene Position.
268      */
269     void lcd_write_char(uint8_t pos, uint8_t ch) {
270         lcd_set_position(pos);
271
272         // Waehle Controller entsprechend Cursorposition
273         if (pos < 54) {
274             lcd_write(LCD_CTL_TOP, LCD_DR, ch);
275         } else {
276             lcd_write(LCD_CTL_BOTTOM, LCD_DR, ch);
277         }
278
279     }
280
281     /*
282      * Schreibt den uebergebenen String an die angegebene Position.
283      */
284     void lcd_write_string(uint8_t pos, char str[]) {
285         uint8_t i = 0;
286
287         // Schreibe das i-te Zeichen an die (pos + i)-te Position.
288         // 0x00 markiert das Ende einer Zeichenkette.
289         while (str[i] != 0) {
290             lcd_write_char(pos + i, str[i]);
291             i++;
292         }
293
294     }
295
296     /*
297      * Leert das LCD.
298      */
299     void lcd_clear(void) {
300         lcd_write(LCD_CTL_TOP, LCD_IR, 0x01);
301         lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x01);
```

```
302 }
303
304 // Aktiviert oder deaktiviert den Cursor.
305 void lcd_display_cursor(uint8_t enable) {
306
307     if (enable) {
308         // Auswahl des Controllers, in dem der Cursor aktiviert
309         // werden soll:
310         if (lcd_cursor & 0x02) {
311
312             // unterer Controller
313             lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x0e);
314             lcd_cursor = 0x03;
315
316         } else {
317
318             // oberer Controller
319             lcd_write(LCD_CTL_TOP, LCD_IR, 0x0e);
320             lcd_cursor = 0x01;
321
322         }
323
324     } else {
325         // Auswahl des Controllers, in dem der Cursor
326         // deaktiviert werden soll:
327
328         if (lcd_cursor & 0x02) {
329
330             // unterer Controller
331             lcd_write(LCD_CTL_BOTTOM, LCD_IR, 0x0c);
332             lcd_cursor = 0x02;
333         } else {
334
335             // oberer Controller
336             lcd_write(LCD_CTL_TOP, LCD_IR, 0x0c);
337             lcd_cursor = 0x00;
338         }
339
340     }
341
342 }
```

A.1.7 eprom.h

```
1 #ifndef __EPROM_H__
2 #define __EPROM_H__
3
4 #include <avr/io.h>
5 #include <inttypes.h>
6 #include <util/delay.h>
```

```
7  #include "misc.h"
8
9  // Konstanten, mit denen der Typ des EPROMs angegeben wird.
10 #define EPROM_NOEPROM 0
11 #define EPROM_NEPROM 1
12 #define EPROM_EEPROM 2
13
14 // Konstante, die die Groesse des EPROMS angibt.
15 #define EPROM_SIZE 2048
16
17 // Konstanten, mit denen angegeben wird, ob das EPROM leer
18 // oder beschrieben ist
19 #define EPROM_NEMPTY 0
20 #define EPROM_EMPTY 1
21
22 uint8_t eprom_get_type(void);
23
24 uint8_t eprom_read(uint16_t address);
25
26 void eprom_write_start(void);
27
28 void eprom_write_stop(void);
29
30 void eprom_write(uint16_t address, uint8_t data);
31
32 uint8_t eprom_is_empty(void);
33
34 void eprom_clear_eeprom(void);
35
36 #endif
```

A.1.8 eprom.c

```
1  #include "eprom.h"
2
3  /*
4   * Ermittelt den EPROM-Typ anhand der Schalterstellung auf der
5   * Platine.
6   */
7  uint8_t eprom_get_type(void) {
8
9      // Falls eine 1 an Pin EPEEP anliegt (der Schalter steht
10     // auf EPROM), ist das EPROM ein (normales) EPROM.
11     if (PIND & (1<<EPEEP)) {
12         return EPROM_NEPROM;
13     } else {
14         return EPROM_EEPROM;
15     }
16
17 }
```

```
18
19  /*
20   * Liest den Wert, der an der uebergebenen Adresse im EPROM
21   * gespeichert ist, aus und gibt ihn zurueck.
22   */
23  uint8_t eprom_read(uint16_t address) {
24
25      // Das EPROM soll auf den LCD-/EPROM-Bus schreiben.
26      misc_set_data_direction(MISC_DD_EPROM);
27
28      // Die Adresse wird auf die Adressleitungen gelegt.
29      PORTB = address & 0xff;
30      PORTD &= ~(0x70);
31      PORTD |= (address>>4) & 0x70;
32
33      // Stabilisierung abwarten.
34      _delay_us(10);
35
36      // Wert lesen und zurueckgeben.
37      return PINA;
38  }
39
40  /*
41   * Speichert, ob der Brennvorgang initialisiert wurde und
42   * welcher EPROM-Typ gebrannt wird.
43   */
44  uint8_t eprom_type = EPROM_NOEPROM;
45
46  /*
47   * Initialisiert den Brennvorgang, muss vor dem ersten Aufruf
48   * von eprom_write() aufgerufen werden.
49   */
50  void eprom_write_start(void) {
51
52      // Falls eprom_write_start() bereits aufgerufen wurde
53      // (ohne zwischenzeitlich eprom_write_stop() aufzurufen),
54      // erzeuge eine Fehlermeldung.
55      if (eprom_type != EPROM_NOEPROM) {
56          misc_error();
57          return;
58      }
59
60      if (eprom_get_type() == EPROM_NEPROM) {
61
62          // Setze den Typ des EPROMs und aktiviere die
63          // Brennspannung fuer das EPROM via VPPWE.
64          eprom_type = EPROM_NEPROM;
65          PORTD &= ~(1<<VPPWE);
66
```

```
67      } else {
68
69          // Setze PROM-Typ auf EPROM, keine Beeinflussung der
70          // Brennspannung noetig
71          eprom_type = EPROM_EEPROM;
72
73      }
74
75      // Warte, bis die Brennspannung korrekt anliegt.
76      _delay_ms(250);
77
78  }
79
80  /*
81   * Beendet den Brennvorgang, sollte nach dem letzten Aufruf von
82   * eprom_write() aufgerufen werden.
83   */
84  void eprom_write_stop(void) {
85
86      switch (eprom_type) {
87
88          case EPROM_NEPROM: // EPROM
89
90              // Deaktiviere Brennspannung und setze den Typ des
91              // PROMs zurueck.
92              PORTD |= (1<<VPPWE);
93              eprom_type = EPROM_NOEPROM;
94              break;
95
96          case EPROM_EEPROM: // EEPROM
97
98              // Setze den Typ des PROMs zurueck.
99              eprom_type = EPROM_NOEPROM;
100             break;
101
102         // eprom_write_stop() wurde aufgerufen, ohne dass
103         // vorher eprom_write_start() aufgerufen wurde.
104         default:
105             misc_error();
106             break;
107
108     }
109
110 }
111
112 /*
113  * Brennt den uebergebenen Wert an die uebergebene Adresse,
114  * eprom_write_start() muss vor dem ersten Aufruf aufgerufen
115  * werden.
```

```
116  */
117 void eprom_write(uint16_t address, uint8_t data) {
118
119     // Der Controller soll auf den LCD-/EPROM-Bus schreiben.
120     misc_set_data_direction(MISC_DD_UC);
121
122     // Die Adresse wird auf die Adress- und der Wert auf die
123     // Datenleitungen gelegt.
124     PORTB = address & 0xff;
125     PORTD &= ~(0x70);
126     PORTD |= (address>>4) & 0x70;
127     PORTA = data;
128
129     // Stabilisierung abwarten.
130     _delay_us(10);
131
132     switch (eprom_type) {
133
134         case EPROM_NEPROM: //EPROM
135
136             // Brennimpuls fuer 50ms anlegen.
137             PORTC |= (1<<EPCE);
138             _delay_ms(50);
139             PORTC &= ~(1<<EPCE);
140             break;
141
142         case EPROM_EEPROM: // EEPROM
143
144             // Brennimpuls geben und Stabilisierung abwarten.
145             PORTD &= ~(1<<VPPWE);
146             _delay_us(10);
147             PORTD |= (1<<VPPWE);
148             break;
149
150         // eprom_write() wurde aufgerufen, ohne dass
151         // vorher eprom_write_start() aufgerufen wurde.
152         default:
153             misc_error();
154             break;
155
156     }
157
158 }
159
160 /*
161  * Liest das EPROM aus und gibt zurueck, ob das EPROM leer ist.
162  */
163 uint8_t eprom_is_empty(void) {
164     uint16_t i = 0;
```

```
165
166     // Pruefe jedes Byte des EPROMS.
167     for (i = 0; i < EPROM_SIZE; i++) {
168
169         // Falls das gelesene Byte nicht 0xFF ist, ist das
170         // EPROM nicht leer.
171         if(eprom_read(i) != 0xff) {
172             break;
173         }
174
175     }
176
177     // Die obige Schleife wird verlassen, wenn eine
178     // beschriebene Speicherzelle gefunden wird oder sie das
179     // ganze PROM abgesucht hat. Durch Vergleich der Ausstiegs-
180     // stelle mit der Groesse erkennt man beschriebene PROMs.
181     if (i < EPROM_SIZE) {
182         return EPROM_NEMPTY;
183     } else { // Das EPROM ist leer
184         return EPROM_EMPTY;
185     }
186
187 }
188
189 /*
190  * Leert das EEPROM, indem in jedes Byte 0xFF geschrieben wird.
191  */
192 void eprom_clear_eeprom(void) {
193
194     // Falls das EPROM kein EEPROM ist, erzeuge eine
195     // Fehlermeldung.
196     if (eprom_get_type() != EPROM_EEPROM) {
197         misc_error();
198         return;
199     }
200
201     // Initialisiere den Brennvorgang.
202     eprom_write_start();
203
204     uint16_t i = 0;
205
206     // Schreibe in jedes Byte 0xFF.
207     for (i = 0; i < EPROM_SIZE; i++) {
208         eprom_write(i, 0xff);
209     }
210
211     // Beende den Brennvorgang.
212     eprom_write_stop();
213 }
```

A.1.9 menu.h

```
#ifndef __MENU_H__
#define __MENU_H__

#include <inttypes.h>
#include "lcd.h"
#include "twi.h"
#include "eprom.h"
#include "editor.h"
#include "pc.h"

// Konstanten, die fuer die verschiedenen Menues stehen.
#define MENU_WAIT 0
#define MENU_MAIN 1
#define MENU_INFO_E_EMPTY 2
#define MENU_INFO_E_N_EMPTY 3
#define MENU_INFO_EE_EMPTY 4
#define MENU_INFO_EE_N_EMPTY 5
#define MENU_INFO_CLEARING_EE 6
#define MENU_PC 7
#define MENU_EDITOR 8
#define MENU_BURN 9
#define MENU_BURNING 10
#define MENU_BURNED 11

// Hier werden die vom Tastencontroller uebergebenen Codes
// menschenlesbaren Konstanten zugewiesen.

#define KEY_0 0x00
#define KEY_1 0x01
#define KEY_2 0x02
#define KEY_3 0x03
#define KEY_4 0x04
#define KEY_5 0x05
#define KEY_6 0x06
#define KEY_7 0x07
#define KEY_8 0x08
#define KEY_9 0x09
#define KEY_A 0x0a
#define KEY_B 0x0b
#define KEY_C 0x0c
#define KEY_D 0x0d
#define KEY_E 0x0e
#define KEY_F 0x0f
#define KEY_LEFT 0x10
#define KEY_RIGHT 0x11
#define KEY_DOWN 0x12
#define KEY_UP 0x13
```

```
48 #define KEY_SK1 0x14
49 #define KEY_SK2 0x15
50 #define KEY_SK3 0x16
51 #define KEY_SK4 0x17
52 #define KEY_OK 0x18
53 #define KEY_PDOWN 0x19
54 #define KEY_PUP 0x1a
55 #define KEY_NONE 0xff
56
57 uint8_t menu_main(void);
58
59 #endif
```

A.1.10 menu.c

```
1  #include "menu.h"
2
3  /*
4   * Zeigt das uebergebene Menue an, indem das LCD erst geleert
5   * und danach neu beschrieben wird.
6   */
7  void menu_display(uint8_t menu) {
8
9      // Leert das LCD, so dass spaeter nur noetige Zeichen
10     // geschrieben werden muessen.
11     lcd_clear();
12
13     switch (menu) {
14
15         case MENU_WAIT: // Warteanzeige
16             lcd_write_string(LCD_LINE1, "Bitte␣warten");
17             lcd_write_string(LCD_LINE4 + 20, "Abbruch");
18             break;
19
20         case MENU_MAIN: // Hauptmenue
21             lcd_write_string(LCD_LINE1, "Hauptmenue");
22             lcd_write_string(LCD_LINE4,
23                     "Editor␣Auslesen␣Info␣␣␣PC");
24             break;
25
26         case MENU_INFO_E_EMPTY: // EPROM ist leer
27             lcd_write_string(LCD_LINE1, "Info");
28             lcd_write_string(LCD_LINE2, "Das␣EPROM␣ist␣leer");
29             lcd_write_string(LCD_LINE4, "reload");
30             lcd_write_string(LCD_LINE4 + 20, "zurueck");
31             break;
32
33         case MENU_INFO_E_N_EMPTY: // EPROM ist nicht leer
34             lcd_write_string(LCD_LINE1, "Info");
35             lcd_write_string(LCD_LINE2,
```

```
36                      "Das␣EPROM␣ist␣nicht␣leer");
37              lcd_write_string(LCD_LINE4, "reload");
38              lcd_write_string(LCD_LINE4 + 20, "zurueck");
39              break;
40
41          case MENU_INFO_EE_EMPTY: // EEPROM ist leer
42              lcd_write_string(LCD_LINE1, "Info");
43              lcd_write_string(LCD_LINE2, "Das␣EEPROM␣ist␣leer");
44              lcd_write_string(LCD_LINE4, "reload");
45              lcd_write_string(LCD_LINE4 + 20, "zurueck");
46              break;
47
48          case MENU_INFO_EE_N_EMPTY: // EEPROM ist nicht leer
49              lcd_write_string(LCD_LINE1, "Info");
50              lcd_write_string(LCD_LINE2,
51                      "Das␣EEPROM␣ist␣nicht␣leer");
52              lcd_write_string(LCD_LINE4, "reload␣loeschen");
53              lcd_write_string(LCD_LINE4 + 20, "zurueck");
54              break;
55
56          case MENU_INFO_CLEARING_EE: // EEPROM wird geloescht
57              lcd_write_string(LCD_LINE1, "Info");
58              lcd_write_string(LCD_LINE2,
59                      "Das␣EEPROM␣wird␣geloescht");
60              break;
61
62          case MENU_PC: // PC-Menue
63              lcd_write_string(LCD_LINE1, "PC");
64              lcd_write_string(LCD_LINE4 + 20, "zurueck");
65              break;
66
67          // Menuezeile des Editors (den Rest schreibt der Editor
68          // selbst):
69          case MENU_EDITOR:
70              lcd_write_string(LCD_LINE4, "Leeren␣Brennen");
71              lcd_write_string(LCD_LINE4 + 20, "zurueck");
72              break;
73
74          case MENU_BURN: // Brenn-Bestaetigung
75              lcd_write_string(LCD_LINE1, "Brennen");
76              lcd_write_string(LCD_LINE2,
77                      "Zum␣Brennen␣OK␣druecken");
78              lcd_write_string(LCD_LINE4 + 20, "Abbruch");
79              break;
80
81          case MENU_BURNING: // EPROM wird gebrannt
82              lcd_write_string(LCD_LINE1, "Brennen");
83              lcd_write_string(LCD_LINE2,
84                      "Das␣EPROM␣wird␣gebrannt");
```

```
85              break;
86
87          case MENU_BURNED: // EPROM wurde gebrannt
88              lcd_write_string(LCD_LINE1, "Brennen");
89              lcd_write_string(LCD_LINE2,
90                      "Das␣EPROM␣wurde␣gebrannt");
91              lcd_write_string(LCD_LINE4 + 20, "zurueck");
92              break;
93
94          default: // Kein gueltiges Menue
95              misc_error();
96              break;
97
98      }
99
100 }
101
102 /*
103  * Wartet auf einen Tastendruck und gibt diesen zurueck.
104  * Zwischen den Tasten-Abfragen wird das PC-Protokoll
105  * abgearbeitet.
106  */
107 uint8_t menu_get_keystroke(void) {
108     uint8_t key = KEY_NONE;
109     uint8_t i = 0;
110
111     // Erst beenden, wenn eine Taste gedrueckt wurde.
112     while (key == KEY_NONE) {
113
114         // Nur alle 10ms Tastendruecke abfragen, in der
115         // Zwischenzeit das PC-Protokoll abarbeiten.
116         for (i = 0; i < 10; i++) {
117             pc_process();
118             _delay_ms(1);
119         }
120
121         // Falls ein neuer Tastendruck anliegt, die Taste lesen
122         // (Ist nur im Fehlerfall false, da bei keiner
123         // gedrueckten Taste die Taste KEY_NONE uebertragen
124         // wird).
125         if (twi_read(2) == TWI_GET_DATA) {
126             key = twi_get_data();
127         }
128
129     }
130
131     return key;
132 }
133
```

```
134 /*
135  * Zeigt das Info-Menue an und ermoegicht ein EEPORM zu leeren.
136  */
137 uint8_t menu_info(void) {
138     uint8_t menu_ret = 0;
139 
140     // Speichert, ob ein nicht-leeres EEPROM eingesteckt ist.
141     uint8_t ee_n_empty = 0;
142 
143     while (menu_ret == 0 || menu_ret == 0xff) {
144 
145         if (menu_ret < 0xff) {
146 
147             // Warte-Anzeige
148             menu_display(MENU_WAIT);
149 
150             if (eprom_is_empty()) { // Leeres (E)EPROM
151 
152                 if (eprom_get_type() == EPROM_NEPROM) {
153                         // Leeres EPROM
154                     menu_display(MENU_INFO_E_EMPTY);
155                 } else { // Leeres EEPROM
156                     menu_display(MENU_INFO_EE_EMPTY);
157                 }
158 
159             } else { // Nicht-leeres (E)EPROM
160 
161                 if (eprom_get_type() == EPROM_NEPROM) {
162                         // Nicht-leeres EPROM
163                     menu_display(MENU_INFO_E_N_EMPTY);
164                 } else { // Nicht-leeres EEPROM
165                     menu_display(MENU_INFO_EE_N_EMPTY);
166                     ee_n_empty = 1;
167                 }
168 
169             }
170 
171             menu_ret = 0xff;
172         }
173 
174         switch (menu_get_keystroke()) {
175 
176             case KEY_SK1: // Aktualisieren
177                 menu_ret = 0;
178                 break;
179 
180             case KEY_SK2: // EEPROM loeschen
181 
182                 // Nur nicht-leere EEPROMs loeschen.
```

```
183                     if (ee_n_empty) {
184                         menu_display(MENU_INFO_CLEARING_EE);
185                         eprom_clear_eeprom();
186                     }
187 
188                     menu_ret = 0;
189                     break;
190 
191                 case KEY_SK4: // Zurueck
192                     menu_ret = 1;
193                     break;
194 
195         }
196 
197     }
198 
199     return menu_ret - 1;
200 }
201 
202 /*
203  * Zeigt das PC-Menue und startet bzw. stoppt das PC-Protokoll.
204  */
205 uint8_t menu_pc(void) {
206     uint8_t menu_ret = 0;
207 
208     while (menu_ret == 0 || menu_ret == 0xff) {
209 
210         if (menu_ret < 0xff) {
211             menu_display(MENU_PC);
212 
213             // Starte das PC-Protokoll.
214             pc_start();
215 
216             menu_ret = 0xff;
217         }
218 
219         switch (menu_get_keystroke()) {
220 
221             case KEY_SK4: // Zurueck
222 
223                 // Stoppt das PC-Protokoll.
224                 pc_stop();
225 
226                 menu_ret = 1;
227                 break;
228 
229         }
230 
231     }
```

```
232
233     return menu_ret - 1;
234 }
235
236 /*
237  * Zeigt das Brennmenue und brennt das EPROM.
238  */
239 uint8_t menu_burning(void) {
240     uint8_t menu_ret = 0;
241
242     while (menu_ret == 0 || menu_ret == 0xff) {
243
244         if (menu_ret < 0xff) {
245
246             // Zeigt die Brennmeldung an, brennt das EPROM und
247             // zeigt abschliessend die Gebrannt-Meldung an.
248             menu_display(MENU_BURNING);
249             editor_burn();
250             menu_display(MENU_BURNED);
251
252             menu_ret = 0xff;
253         }
254
255         switch (menu_get_keystroke()) {
256
257             case KEY_SK4: // Zurueck zum Editor
258                 menu_ret = 2;
259                 break;
260
261         }
262
263     }
264
265     return menu_ret - 1;
266 }
267
268 /*
269  * Zeigt das Brennbestaetigungsmenue.
270  */
271 uint8_t menu_burn(void) {
272     uint8_t menu_ret = 0;
273
274     while (menu_ret == 0 || menu_ret == 0xff) {
275
276         if (menu_ret < 0xff) {
277             menu_display(MENU_BURN);
278             menu_ret = 0xff;
279         }
280
```

```
281         switch (menu_get_keystroke()) {
282 
283             case KEY_OK: // Brennen
284                 menu_ret = menu_burning();
285                 break;
286 
287             case KEY_SK4: // Zurueck
288                 menu_ret = 1;
289                 break;
290 
291         }
292 
293     }
294 
295     return menu_ret - 1;
296 }
297 
298 /*
299  * Zeigt den Editor an.
300  */
301 uint8_t menu_editor(void) {
302     uint8_t menu_ret = 0;
303     uint8_t key = KEY_NONE;
304 
305     while (menu_ret == 0 || menu_ret == 0xff) {
306 
307         if (menu_ret < 0xff) {
308             menu_display(MENU_EDITOR);
309 
310             // Zeigt Cursor und Editor an.
311             lcd_display_cursor(1);
312             editor_display();
313 
314             menu_ret = 0xff;
315         }
316 
317         key = menu_get_keystroke();
318 
319         switch (key) {
320 
321             case KEY_SK1: // Editor leeren
322                 editor_clear();
323                 menu_ret = 0;
324                 break;
325 
326             case KEY_SK2: // Brennen
327 
328                 // Cursor abschalten.
329                 lcd_display_cursor(0);
```

```
330                     menu_ret = menu_burn();
331                     break;
332 
333                 case KEY_SK3: // Unbenutzt
334                     break;
335 
336                 case KEY_SK4: // Zurueck
337                     menu_ret = 1;
338                     break;
339 
340                 default: // Editortaste wurde gedrueckt
341                     editor_handle_keystroke(key);
342                     break;
343 
344         }
345 
346     }
347 
348     // Cursor abschalten.
349     lcd_display_cursor(0);
350     return menu_ret - 1;
351 }
352 
353 /*
354  * Zeigt das Hauptmenue in einer Endlosschleife an und wartet
355  * auf einen Tastendruck. Wird ein Softkey gedrueckt, wird das
356  * entsprechende Untermenue aufgerufen.
357  */
358 uint8_t menu_main(void) {
359     uint8_t menu_ret = 0;
360 
361     while (1) {
362 
363         if (menu_ret < 0xff) {
364             menu_display(MENU_MAIN);
365             menu_ret = 0xff;
366         }
367 
368         switch (menu_get_keystroke()) {
369 
370             case KEY_SK1: // Editor
371                 menu_ret = menu_editor();
372                 break;
373 
374             case KEY_SK2: // Auslesen
375                 menu_display(MENU_WAIT);
376                 editor_read_eprom();
377                 menu_ret = menu_editor();
378                 break;
```

```
379
380                 case KEY_SK3: // Info
381                     menu_ret = menu_info();
382                     break;
383
384                 case KEY_SK4: // PC
385                     menu_ret = menu_pc();
386                     break;
387
388             }
389
390         }
391
392         return 0;
393 }
```

A.1.11 editor.h

```
1  #ifndef __EDITOR_H__
2  #define __EDITOR_H__
3
4  #include "lcd.h"
5  #include "menu.h"
6  #include "misc.h"
7
8  // Konstante, die die Speichergroesse des Editors angibt.
9  #define EDITOR_MAXMEMSIZE EPROM_SIZE
10
11 uint16_t editor_end_pointer;
12 uint8_t editor_buffer[EDITOR_MAXMEMSIZE];
13
14 void editor_init(void);
15 void editor_handle_keystroke(uint8_t key);
16 void editor_display(void);
17 void editor_clear(void);
18 void editor_read_eprom(void);
19 void editor_burn(void);
20
21 #endif
```

A.1.12 editor.c

```
1  #include "editor.h"
2
3  // Addresse des ersten angezeigten Bytes (links oben im Display)
4  uint16_t editor_start = 0;
5
6  // Zaehlt Position der Halbbytes
7  uint8_t editor_cursor_position = 0;
8
```

```
// Adresse nach dem letzten bearbeiteten Byte
uint16_t editor_end_pointer = 0;

// Zwischenspeicher fuer eingegebene Daten
uint8_t editor_buffer[EDITOR_MAXMEMSIZE];

// Sprungziele fuer den Cursor; Beginn der einzelnen Bytes
const uint8_t editor_tabs[] = {7, 9, 12, 14, 18, 20, 23, 25};

// Sprungziele fuer den Cursor; Beginn der einzelnen Half-Bytes
const uint8_t editor_real_pos[] = {7, 8, 9, 10, 12, 13, 14, 15,
        18, 19, 20, 21, 23, 24, 25, 26};

/*
 * Inititalisiert den Editor mit einem leeren Puffer
 */
void editor_init(void) {
    uint16_t i = 0;

    // Jedes Zeichen im Puffer auf 0xff setzen (leeren)
    for (i = 0; i < EDITOR_MAXMEMSIZE; i++) {
        editor_buffer[i] = 0xff;
    }

}

/*
 * Setzt den Cursor des LCD an die Position, auf die
 * editor_cursor zeigt.
 */
void editor_update_cursor(void) {

    // Berechnet die Zeile und die Position in der Zeile und
    // setzt den Cursor dorthin.
    lcd_set_position(editor_cursor_position / 16 * 27 +
            editor_real_pos[editor_cursor_position % 16]);
}

/*
 * Zeichnet den Editorinhalt der aktuellen Stelle.
 */
void editor_display(void) {

    // Speichert LCD-Zeilen-Konstanten fuer einfachen Zugriff
    uint8_t line[] = {LCD_LINE1, LCD_LINE2, LCD_LINE3,
            LCD_LINE4};

    uint8_t j = 0;

```

```
58      // Zeichne Zeilen 1-3 (in der 4. stehen die Menuepunkte)
59      for (j = 0; j < 3; j++) {
60
61          // Initialisiere eine Zeile der Editoransicht und
62          // trage Adresse (links) und Daten (rechts) ein
63          char str[28] = "␣␣␣h:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";
64          str[0] = misc_halfbyte_to_char(
65                  (editor_start + j * 8)>>8);
66          str[1] = misc_halfbyte_to_char(
67                  (editor_start + j * 8)>>4);
68          str[2] = misc_halfbyte_to_char(
69                  editor_start + j * 8);
70
71          uint8_t i = 0;
72
73          for (i = 0; i < 8; i++) {
74              str[editor_tabs[i]] = misc_halfbyte_to_char(
75                      editor_buffer[editor_start + j * 8 + i]>>4);
76              str[editor_tabs[i] + 1] = misc_halfbyte_to_char(
77                      editor_buffer[editor_start + j * 8 + i]);
78          }
79
80          // Gibt den String im LCD aus.
81          lcd_write_string(line[j], str);
82      }
83
84      // Setzt den Cursor
85      editor_update_cursor();
86  }
87
88  /*
89   * Wertet Tasteneingaben aus.
90   */
91  void editor_handle_keystroke(uint8_t key) {
92
93      switch (key) {
94
95          // Je nach Cursorposition: Ein Halfbyte nach links
96          // oder eine Zeile nach oben und ganz nach rechts
97          case KEY_LEFT:
98
99              if (editor_cursor_position != 0) {
100                 // Cursor ist noch nicht links oben, also
101                 // Cursor eins nach vorne und anzeigen.
102                 editor_cursor_position--;
103                 editor_update_cursor();
104             } else if (editor_start != 0) {
105                 // Cursor ist links oben und es gibt noch
106                 // Zeilen weiter oben. Dann: Eine Zeile nach
```

```
107                 // oben scrollen und Cursor aktualisieren.
108                 editor_start = editor_start - 8;
109                 editor_cursor_position = 15;
110                 editor_display();
111             }
112 
113             break;
114 
115         // Je nach Cursorposition: ein Halfbyte nach rechts
116         // oder herunterscrollen und Cursor an den Anfang der
117         // nun letzten Zeile setzen.
118         case KEY_RIGHT:
119 
120             if (editor_cursor_position != 47) {
121                 // Nicht rechts unten: Cursor eins nach hinten
122                 editor_cursor_position++;
123                 editor_update_cursor();
124             } else if (EDITOR_MAXMEMSIZE - 24 > editor_start) {
125                  // Cursor ist rechts unten und es gibt noch
126                  // Zeilen weiter unten: Eine Zeile nach unten
127                  // scrollen, Cursor setzen und aktualisieren.
128                 editor_start = editor_start + 8;
129                 editor_cursor_position = 32;
130                 editor_display();
131             }
132 
133             break;
134 
135         // Cursor eine Zeile nach oben, ggf. nach oben scrollen.
136         case KEY_UP:
137 
138             if (editor_cursor_position / 16 > 0) {
139                 // Cursor ist nicht in oberster Zeile:
140                 // Cursor setzen und anzeigen
141                 editor_cursor_position =
142                     editor_cursor_position - 16;
143                 editor_update_cursor();
144             } else if (editor_start > 0) {
145                 // Cursor ist in oberster Zeile und es gibt
146                 // noch Zeilen weiter oben. Dann: Hochscrollen.
147                 editor_start = editor_start - 8;
148                 editor_display();
149             }
150 
151             break;
152 
153         // Eine Zeile nach unten, ggf. nach unten scrollen.
154         case KEY_DOWN:
155 
```

```
156             if (editor_cursor_position / 16 < 2) {
157                 // Cursor ist nicht in unterster Zeile:
158                 // Cursor setzen und anzeigen
159                 editor_cursor_position = editor_cursor_position
160                         + 16;
161                 editor_update_cursor();
162             } else if (EDITOR_MAXMEMSIZE - 24 > editor_start) {
163                 // Cursor ist in unterster Zeile und es gibt noch
164                 // Zeilen weiter unten. Dann: Herunterscrollen.
165                 editor_start = editor_start + 8;
166                 editor_display();
167             }
168 
169             break;
170 
171         // Zeichen in LCD und den Puffer schreiben und Cursor
172         // weiterruecken.
173         case KEY_0:
174         case KEY_1:
175         case KEY_2:
176         case KEY_3:
177         case KEY_4:
178         case KEY_5:
179         case KEY_6:
180         case KEY_7:
181         case KEY_8:
182         case KEY_9:
183         case KEY_A:
184         case KEY_B:
185         case KEY_C:
186         case KEY_D:
187         case KEY_E:
188         case KEY_F:
189 
190             {
191                 // Aktuelle Position im Puffer
192                 uint16_t pos = editor_start +
193                         editor_cursor_position / 2;
194 
195                 if (editor_cursor_position % 2 == 0) {
196                     // Linkes Half-Byte speichern
197                     editor_buffer[pos] = (key<<4) |
198                             (editor_buffer[pos] & 0x0f);
199                 } else {
200                     // Rechtes Half-Byte speichern
201                     editor_buffer[pos] = key |
202                             (editor_buffer[pos] & 0xf0);
203                 }
204 
```

```
205                 // Endpointer neu setzen, wenn ein anderes Zeichen
206                 // als 0xff hinter dem aktuellen Endpointer
207                 // geschrieben wurde.
208                 if (key != KEY_F && pos + 1 >
209                         editor_end_pointer) {
210                     editor_end_pointer = pos + 1;
211                 }
212 
213                 // Zeichen auf LCD schreiben
214                 lcd_write_char(editor_cursor_position / 16 * 27
215                         + editor_real_pos[editor_cursor_position
216                         % 16], misc_halfbyte_to_char(key));
217 
218                 // Eine Position weiterruecken
219                 editor_handle_keystroke(KEY_RIGHT);
220                 break;
221             }
222 
223         // Eine Editorseite (drei Zeilen) nach unten scrollen.
224         case KEY_PDOWN:
225 
226             // Je nach Anzahl der noch vorhandenen Zeilen im
227             // Puffer versuchen, drei oder die max. noch
228             // vorhandene Zeilenzahl herunterzuscrollen
229             if (EDITOR_MAXMEMSIZE - 40 > editor_start) {
230                 editor_start = editor_start + 24;
231                 editor_display();
232             } else if (EDITOR_MAXMEMSIZE - 32 > editor_start) {
233                 editor_start = editor_start + 16;
234                 editor_display();
235             } else if (EDITOR_MAXMEMSIZE - 24 > editor_start) {
236                 editor_start = editor_start + 8;
237                 editor_display();
238             }
239 
240             break;
241 
242         // Eine Editorseite (drei Zeilen) nach unten scrollen.
243         case KEY_PUP:
244 
245             // Je nach Anzahl der noch vorhandenen Zeilen im
246             // Puffer versuchen, drei oder die max. noch
247             // vorhandene Zeilenzahl hinaufzuscrollen
248             if (editor_start > 16) {
249                 editor_start = editor_start - 24;
250                 editor_display();
251             } else if (editor_start > 8) {
252                 editor_start = editor_start - 16;
253                 editor_display();
```

```
254                 } else if (editor_start > 0) {
255                     editor_start = editor_start - 8;
256                     editor_display();
257                 }
258 
259                 break;
260 
261             // Unbekannte Taste, nichts tun.
262             default:
263                 break;
264 
265     }
266 
267 }
268 
269 /*
270  * Editor leeren und alle Variablen zuruecksetzen.
271  */
272 void editor_clear(void) {
273     uint16_t i = 0;
274 
275     // Puffer bis zum End Pointer (nach dem nur noch 0xFF
276     // folgt) wieder mit 0xFF fuellen
277     for (i = 0; i < editor_end_pointer; i++) {
278         editor_buffer[i] = 0xff;
279     }
280 
281     // Alles zuruecksetzen.
282     editor_end_pointer = 0;
283     editor_start = 0;
284     editor_cursor_position = 0;
285 }
286 
287 /*
288  * EPROM in den Puffer lesen.
289  */
290 void editor_read_eprom(void) {
291     editor_end_pointer = 0;
292     uint16_t i = 0;
293 
294     // EPROM auslesen und Endpointer neu setzen
295     for (i = 0; i < EDITOR_MAXMEMSIZE; i++) {
296         editor_buffer[i] = eprom_read(i);
297 
298         if (editor_buffer[i] != 0xff) {
299             editor_end_pointer = i + 1;
300         }
301 
302     }
```

```
303
304     // Anzeige auf Anfang setzen.
305     editor_start = 0;
306     editor_cursor_position = 0;
307 }
308
309 /*
310  * Brennt den Inhalt des Puffers in das EPROM.
311  */
312 void editor_burn(void) {
313
314     // Brennvorgang initialisieren.
315     eprom_write_start();
316     uint16_t i = 0;
317
318     // EPROM brennen, dabei 0xff auslassen.
319     for (i = 0; i < editor_end_pointer; i++) {
320
321         if (editor_buffer[i] != 0xff) {
322             eprom_write(i, editor_buffer[i]);
323         }
324
325     }
326
327     // Brennvorgang beenden.
328     eprom_write_stop();
329 }
```

A.1.13 twi.h

```
1  #ifndef __TWI_H__
2  #define __TWI_H__
3
4  #include <avr/io.h>
5  #include <inttypes.h>
6
7  // Konstanten, die die eigene Adresse und das Timeout angeben.
8  #define TWI_ADDRESS 1
9  #define TWI_TIMEOUT 2000
10
11 // Konstanten, die fuer die Zustaende stehen.
12 #define TWI_STATE_STOP 0
13 #define TWI_STATE_START 1
14 #define TWI_STATE_ADDRESS_SET 2
15 #define TWI_STATE_ADDRESS 3
16 #define TWI_STATE_READ 4
17 #define TWI_STATE_READ_GET 5
18
19 // Konstanten, die fuer Statusmeldunden stehen.
20 #define TWI_NOP 0
```

```
21 #define TWI_SET_ADDRESS 1
22 #define TWI_GET_DATA 2
23 #define TWI_ERROR 3
24
25 void twi_init(void);
26
27 uint8_t twi_get_data(void);
28
29 uint8_t twi_read(uint8_t address);
30
31 #endif
```

A.1.14 twi.c

```
1  #include "twi.h"
2
3  // Speichert den aktuellen Zustand.
4  uint8_t twi_state = TWI_STATE_STOP;
5
6  // Speichert den zuletzt uebertragenen Wert
7  uint8_t twi_data = 0xff;
8
9  /*
10  * Initialisiert TWI. Setzt die eigene Adresse sowie die
11  * Uebertragungsgeschwindigkeit und aktiviert TWI.
12  */
13 void twi_init(void) {
14
15     // Setze die eigene Adresse.
16     TWAR = TWI_ADDRESS<<1;
17
18     // Setze die Uebertragungsgeschwindigkeit
19     TWBR = 32;
20     TWSR = 2;
21
22     // Aktiviere TWI.
23     TWCR = (1<<TWEN);
24 }
25
26 /*
27  * Setzt die Slave-Adresse und sendet diese.
28  */
29 void twi_set_address(uint8_t address) {
30
31     // Nur senden, wenn im richtigen Zustand.
32     if (twi_state == TWI_STATE_ADDRESS_SET) {
33
34         // In neuen Zustand gehen
35         twi_state = TWI_STATE_ADDRESS;
36
```

```
37          // Adresse setzen und ReceiverMode aktivieren.
38          TWDR = (address<<1) | 0x01;
39
40          // Uebertragen.
41          TWCR = (1<<TWEN) | (1<<TWINT);
42      }
43  }
44
45  /*
46   * Startet die Uebertragung.
47   */
48  void twi_start(void) {
49
50      // Nur starten, wenn im richtigen Zustand.
51      if (twi_state == TWI_STATE_STOP) {
52          twi_state = TWI_STATE_START;
53
54          // Start senden.
55          TWCR = (1<<TWEN) | (1<<TWSTA);
56      }
57  }
58
59  /*
60   * Stoppt die Uebertragung unabhaengig vom Zustand.
61   */
62  void twi_stop(void) {
63      twi_state = TWI_STATE_STOP;
64
65      // Stop senden, Uebertragung beenden.
66      TWCR = (1<<TWEN) | (1<<TWINT) | (1<<TWSTO);
67  }
68
69  /*
70   * Gibt den zuletzt empfangenen Wert zurueck.
71   */
72  uint8_t twi_get_data(void) {
73
74      // Nur im richtigen Zustand Uebertragung beenden.
75      if (twi_state == TWI_STATE_READ_GET) {
76          twi_stop();
77      }
78
79      // Wert zurueckgeben.
80      return twi_data;
81  }
82
83  /*
84   * Prueft, ob sich bei der Uebertragung etwas getan hat und
85   * reagiert darauf.
```

```
86    */
87   uint8_t twi_process(void) {
88
89       // Interrupt ist aufgetreten
90       if (TWCR & (1<<TWINT)) {
91
92           // Status auslesen und verarbeiten
93           switch (TWSR & 0xf8) {
94
95               // Start wurde uebertragen
96               case 0x08:
97
98                   if (twi_state == TWI_STATE_START) {
99
100                      // Aufrufer mitteilen, dass Adresse gesetzt
101                      // werden kann und Zustand wechseln.
102                      twi_state = TWI_STATE_ADDRESS_SET;
103                      return TWI_SET_ADDRESS;
104
105                  } else if (twi_state == TWI_STATE_ADDRESS_SET) {
106
107                      // Derselbe Interrupt, es wurde nur
108                      // noch nicht die Adresse gesetzt.
109                      // Aufrufer (noch einmal) mitteilen,
110                      // dass Adresse gesetzt werden kann.
111                      return TWI_SET_ADDRESS;
112
113                  } else { // Im falschen Zustand
114
115                      // Uebertragung beenden.
116                      twi_stop();
117                      return TWI_ERROR;
118                  }
119                  break;
120
121              // Bestaetigung vom Slave eingetroffen, dass er
122              // einen Wert senden soll.
123              case 0x40:
124
125                  if (twi_state == TWI_STATE_ADDRESS) {
126
127                      // Wert empfangen, dabei kein ACK senden.
128                      twi_state = TWI_STATE_READ;
129                      TWCR = (1<<TWEN) | (1<<TWINT);
130
131                  } else { // Im falschen Zustand
132
133                      // Uebertragung beenden.
134                      twi_stop();
```

```
135                     return TWI_ERROR;
136                 }
137                 break;
138 
139             // Wert vom Slave empfangen, kein ACK gesendet.
140             case 0x58:
141 
142                 if (twi_state == TWI_STATE_READ) {
143                     twi_state = TWI_STATE_READ_GET;
144                     twi_data = TWDR;
145                     return TWI_GET_DATA;
146                 } else if (twi_state == TWI_STATE_READ_GET) {
147 
148                     // Derselbe Interrupt, Wert noch nicht
149                     // gelesen. Wiederholt senden, dass ein
150                     // Wert gelesen werden kann.
151                     return TWI_GET_DATA;
152 
153                 } else { // Im falschen Zustand
154 
155                     // Uebertragung beenden.
156                     twi_stop();
157                     return TWI_ERROR;
158                 }
159                 break;
160 
161             // Nicht erwarteter Status, Fehler annehmen.
162             default:
163 
164                 // Uebertragung beenden.
165                 twi_stop();
166                 return TWI_ERROR;
167                 break;
168         }
169     }
170 
171     // Mitteilen, dass nichts passiert ist.
172     return TWI_NOP;
173 }
174 
175 /*
176  * Initiert eine Uebertragung und gibt eine Statusmeldung
177  * zurueck.
178  */
179 uint8_t twi_read(uint8_t address) {
180     uint16_t i = 0;
181 
182     // Uebertragung beginnen.
183     twi_start();
```

```
184
185     // Solange kein Timeout vorliegt, weiter arbeiten.
186     while (i < TWI_TIMEOUT) {
187
188         //
189         switch (twi_process()) {
190
191             // Adresse kann gesetzt werden.
192             case TWI_SET_ADDRESS:
193
194                 // Timeout-Zaehler zuruecksetzen.
195                 i = 0;
196
197                 // Adresse setzen und uebertragen.
198                 twi_set_address(address);
199                 break;
200
201             // Wert wurde empfangen, kann abgefragt werden.
202             case TWI_GET_DATA:
203
204                 // Dem Aufrufer mitteilen, dass ein Wert
205                 // angekommen ist.
206                 return TWI_GET_DATA;
207                 break;
208
209             // Noch nichts passiert
210             case TWI_NOP:
211
212                 // Timeout-Zaehler erhoehen.
213                 i++;
214                 break;
215
216             // Fehler oder falscher Wert wurde zurueckgegeben
217             default:
218
219                 // Uebertragung stoppen, Fehler zurueckgeben.
220                 twi_stop();
221                 return TWI_ERROR;
222                 break;
223         }
224     }
225
226     // Timeout wurde erreicht, Uebertragung beenden, Fehler
227     // zurueckgeben.
228     twi_stop();
229     return TWI_ERROR;
230 }
```

A.1.15 usart.h

```c
#ifndef __USART_H__
#define __USART_H__

#include <avr/io.h>
#include <inttypes.h>
#include <stdio.h>

void usart_init(uint16_t baud, uint8_t u2x);

uint8_t usart_has_data(void);

uint8_t usart_read_char(void);

void usart_write_char(uint8_t data);

void usart_write_string(uint8_t str[]);

#endif
```

A.1.16 usart.c

```c
#include "usart.h"

/*
 * Sendet ein Zeichen (1 Byte) ueber RS232
 */
int usart_write(char data, FILE *stream) {

    // Warte, bis das letzte Byte komplett versendet wurde.
    while (!(UCSR0A & (1<<UDRE0)));

    // Setze das zu sendende Zeichen.
    UDR0 = data;
    return 0;
}

// Datei-Deskriptor, der als stdout benutzt wird und ueber
// RS232 sendet.
FILE usart_out = FDEV_SETUP_STREAM(usart_write, NULL,
        _FDEV_SETUP_WRITE);

/*
 * Initialisiert RSR232 und setzt stdout zum Senden ueber
 * RS232.
 */
void usart_init(uint16_t baud, uint8_t u2x) {

    // Divisor bei der Berechnung des Wertes des Baud Rate
    // Registers.
    uint16_t divisor = 16;
```

```
30
31      // Wenn die doppelte Baud Rate benutzt wird, muss der
32      // Divisor halbiert werden.
33      if (u2x == 1) {
34          divisor = 8;
35          UCSR0A |= (1<<U2X0);
36      }
37
38      // Berechnen des Wertes des Baud Rate Registers.
39      uint16_t ubrr = F_CPU / divisor / baud - 1;
40
41      // Baud Rate Register setzen.
42      UBRR0H = (ubrr & 0x0f00)>>8;
43      UBRR0L = ubrr & 0x00ff;
44
45      // USART aktivieren.
46      UCSR0B = (1<<RXEN0) | (1<<TXEN0);
47
48      // Setze neuen stdout.
49      stdout = &usart_out;
50  }
51
52  /*
53   * Gibt zurueck, ob ein Zeichen empfangen wurde.
54   */
55  uint8_t usart_has_data(void) {
56
57      if ((UCSR0A & (1<<RXC0))) { // Zeichen empfangen
58          return 1;
59      } else { // Kein Zeichen empfangen
60          return 0;
61      }
62
63  }
64
65  /*
66   * Wartet, bis ein Zeichen empfangen wurde und gibt dies
67   * zurueck. Achtung, wenn kein Zeichen empfangen wird, wird
68   * diese Funktion nie beendet.
69   */
70  uint8_t usart_read_char(void) {
71      while (!usart_has_data());
72      return UDR0;
73  }
74
75  /*
76   * Sendet ein einzelnes Zeichen.
77   */
78  void usart_write_char(uint8_t data) {
```

```
79      usart_write(data, NULL);
80  }
81
82  /*
83   * Sendet einen String, indem es jedes Zeichen einzeln sendet.
84   */
85  void usart_write_string(uint8_t str[]) {
86      uint8_t i = 0;
87
88      // 0x00 beendet einen String, also dann abbrechen.
89      while (str[i] != 0) {
90          usart_write_char(str[i]);
91          i++;
92      }
93
94  }
```

A.1.17 pc.h

```
1   #ifndef __PC_H__
2   #define __PC_H__
3
4   #include <inttypes.h>
5   #include "usart.h"
6   #include "eprom.h"
7   #include "editor.h"
8
9   // Konstanten, die fuer die Zustaende stehen.
10  #define PC_STATE_STOP 0
11  #define PC_STATE_START 1
12  #define PC_STATE_CONNECTED 2
13  #define PC_STATE_WRITE_SIZE 3
14  #define PC_STATE_WRITE 4
15
16  // Konstante, die die Version des Protokolls angibt.
17  #define PC_VERSION 1
18
19  // Konstanten, die fuer die Nachrichten stehen.
20  #define PC_DISCONNECT 0
21  #define PC_CONNECT 1
22  #define PC_SIZE 2
23  #define PC_INFO 3
24  #define PC_NEPROM 4
25  #define PC_EEPROM 5
26  #define PC_EMPTY 6
27  #define PC_NEMPTY 7
28  #define PC_CLEAR 8
29  #define PC_CLEARED 9
30  #define PC_NCLEARED 10
31  #define PC_FLUSH 11
```

```
32 #define PC_FLUSHED 12
33 #define PC_READ 13
34 #define PC_WRITE 14
35 #define PC_WRITE_SIZE 15
36 #define PC_WRITE_NEXT 16
37 #define PC_WRITTEN 17
38 #define PC_BURN 18
39 #define PC_BURNING 19
40 #define PC_BURNED 20
41
42 void pc_start(void);
43
44 void pc_process(void);
45
46 void pc_stop(void);
47
48 #endif
```

A.1.18 pc.c

```
1  #include "pc.h"
2
3  // Speichert den aktuellen Zustand.
4  uint8_t pc_state = PC_STATE_STOP;
5
6  // Speichert die Groesse der zu empfangenen Daten.
7  uint16_t pc_write_size = 0;
8
9  // Speichert die Anzahl der bereits empfangenen Daten.
10 uint16_t pc_write_i = 0;
11
12 // Leert den Empfangspuffer.
13 void pc_flush(void) {
14
15     // Solange noch Daten im Puffer sind, auslesen.
16     while (usart_has_data()) {
17         usart_read_char();
18     }
19 }
20
21 /*
22  * Startet das PC-Protokoll
23  */
24 void pc_start(void) {
25
26     // Falls das PC-Protokoll laeuft, erst stoppen.
27     if (pc_state != PC_STATE_STOP) {
28         pc_stop();
29     }
30
```

```
31      // Puffer leeren und in den Start-Zustand gehen.
32      pc_state = PC_STATE_START;
33      pc_flush();
34  }
35
36  /*
37   * Prueft, ob Daten empfangen wurden und verarbeitet diese.
38   */
39  void pc_process(void) {
40
41      // Laeuft das PC-Protokoll nicht oder wurden noch keine
42      // Daten empfangen, abbrechen.
43      if (pc_state == PC_STATE_STOP || !usart_has_data()) {
44          return;
45      }
46
47      // Empfangenes Zeichen lesen.
48      uint8_t data = usart_read_char();
49
50      // Verarbeitung abhaengig vom aktuellen Zustand.
51      switch (pc_state) {
52
53          case PC_STATE_START: // Start-Zustand
54
55              // Wenn PC_CONNECT empfangen wurde, Version des
56              // Protokolls senden in den Zustand Verbunden
57              // uebergehen.
58              if (data == PC_CONNECT) {
59                  pc_state = PC_STATE_CONNECTED;
60                  usart_write_char(PC_VERSION);
61              }
62
63              break;
64
65          case PC_STATE_CONNECTED: // Zustand Verbunden
66
67              // Auf empfangenes Zeichen reagieren.
68              switch (data) {
69
70                  // Die Verbindung wird abgebaut, alle Variablen
71                  // zuruecksetzen, in den Start-Zustand gehen.
72                  case PC_DISCONNECT:
73                      pc_state = PC_STATE_START;
74                      pc_write_size = 0;
75                      pc_write_i = 0;
76                      usart_write_char(PC_DISCONNECT);
77                      break;
78
79                  // Die maximale Groesse des Editors wird
```

```
80              // angefordert, erst wird das MSB,
81              // anschliessend das LSB gesendet.
82              case PC_SIZE:
83                  usart_write_char(EDITOR_MAXMEMSIZE>>8);
84                  usart_write_char(
85                          (uint8_t) EDITOR_MAXMEMSIZE);
86                  break;
87
88              // Information ueber das EPROM wird
89              // angefordert, erst wird der Typ gesendet,
90              // dann, ob es leer ist.
91              case PC_INFO:
92
93                  if (eprom_get_type() == EPROM_NEPROM) {
94                      usart_write_char(PC_NEPROM);
95                  } else {
96                      usart_write_char(PC_EEPROM);
97                  }
98
99                  if (eprom_is_empty()) {
100                     usart_write_char(PC_EMPTY);
101                 } else {
102                     usart_write_char(PC_NEMPTY);
103                 }
104
105                 break;
106
107             // Das nicht-leere EEPROM soll geloescht
108             // werden. Ist ein nicht-leeres EPROM
109             // eingesteckt, so wird es geloescht und es
110             // wird CLEARED gesendet, ansonsten wird
111             // NCLEARED gesendet.
112             case PC_CLEAR:
113
114                 if (eprom_get_type() == EPROM_EEPROM &&
115                         !eprom_is_empty()) {
116                     eprom_clear_eeprom();
117                     usart_write_char(PC_CLEARED);
118                 } else {
119                     usart_write_char(PC_NCLEARED);
120                 }
121
122                 break;
123
124             // Der Editor soll geleert werden, also wird er
125             // geleert und FLUSHED gesendet.
126             case PC_FLUSH:
127                 editor_clear();
128                 usart_write_char(PC_FLUSHED);
```

```
129                     break;
130
131                 // Der PC fordert die Daten aus dem EPROM an.
132                 case PC_READ:
133
134                     // Das EPROM auslesen.
135                     editor_read_eprom();
136
137                     // Groesse der EPROM-Daten senden,
138                     // MSB zuerst.
139                     usart_write_char(editor_end_pointer>>8);
140
141                     // Stabilisierung abwarten
142                     _delay_ms(60);
143
144                     // LSB uebertragen
145                     usart_write_char(editor_end_pointer);
146
147                     uint16_t i = 0;
148
149                     // Daten uebertragen, dabei jeweils warten.
150                     for (i = 0; i < editor_end_pointer; i++) {
151                         _delay_ms(60);
152                         usart_write_char(editor_buffer[i]);
153                     }
154
155                     break;
156
157                 // PC will in den Editor schreiben -- Zustand
158                 // wechseln und Bestaetigung senden.
159                 case PC_WRITE:
160                     pc_state = PC_STATE_WRITE_SIZE;
161                     usart_write_char(PC_WRITE_SIZE);
162                     break;
163
164                 // Das EPROM soll gebrannt werden, also
165                 // Bestaetigung senden, das EPROM brennen und
166                 // dann erneut Bestaetigung senden.
167                 case PC_BURN:
168                     usart_write_char(PC_BURNING);
169                     editor_burn();
170                     usart_write_char(PC_BURNED);
171                     break;
172             }
173             break;
174
175         // Die Groesse der zu empfangenen Daten wird empfangen.
176         case PC_STATE_WRITE_SIZE:
177
```

```
178             if (pc_write_i == 0) { // MSB zuerst
179
180                 // Zaehlvariable setzen, Datum speichern und
181                 // Bestaetigung senden.
182                 pc_write_i = 1;
183                 pc_write_size = data<<8;
184                 usart_write_char(PC_WRITE_SIZE);
185
186             } else { // Das naechste das LSB
187
188                 // Zaehlvariable zuruecksetzen, Datum speichern
189                 // und Bestaetigung senden sowie Zustand
190                 // wechseln.
191                 pc_write_i = 0;
192                 pc_state = PC_STATE_WRITE;
193                 pc_write_size |= data;
194                 usart_write_char(PC_WRITE_NEXT);
195
196             }
197
198             break;
199
200         // Die Daten fuer den Editor werden empfangen.
201         case PC_STATE_WRITE:
202
203             // Datum speichern, Zaehler erhoehen.
204             editor_buffer[pc_write_i++] = data;
205
206             if (pc_write_i >= pc_write_size) {
207                 // Alle Daten empfangen.
208
209                 // In den Zustand Verbunden wechseln, Zaehler
210                 // zuruecksetzen und Bestaetigung schicken.
211                 pc_state = PC_STATE_CONNECTED;
212                 pc_write_i = 0;
213                 usart_write_char(PC_WRITTEN);
214
215             } else { // Daten stehen noch aus
216
217                 // Bestaetigung senden, weitere Daten erwarten.
218                 usart_write_char(PC_WRITE_NEXT);
219
220             }
221
222             break;
223
224         // Unbekanntes Zeichen verwerfen.
225         default:
226             break;
```

```
227
228     }
229
230 }
231
232 /*
233  * Stoppt das PC-Protokoll und setzt alle Variablen zurueck.
234  */
235 void pc_stop(void) {
236
237     // Nur zuruecksetzen, wenn nicht bereits gestoppt.
238     if (pc_state != PC_STATE_STOP) {
239
240         // Falls noch verbunden, trennen.
241         if (pc_state != PC_STATE_START) {
242             usart_write_char(PC_DISCONNECT);
243         }
244
245         // Zuruecksetzen.
246         pc_state = PC_STATE_STOP;
247         pc_write_size = 0;
248         pc_write_i = 0;
249
250     }
251
252 }
```

A.2 Tastaturcontroller

A.2.1 tastencontroller.h

```
1  #ifndef __TASTENCONTROLLER_H__
2  #define __TASTENCONTROLLER_H__
3
4  #include <avr/io.h>
5  #include <inttypes.h>
6  #include <util/delay.h>
7  #include "usi.h"
8
9  // Konstanten fuer die Zuordnung von Gruppenleitungen der
10 // Tastatur zu Pins an Port D
11 #define WIRE_GROUP1 2
12 #define WIRE_GROUP2 3
13 #define WIRE_GROUP3 4
14 #define WIRE_GROUP4 5
15
16 // Konstanten fuer die Zuordnung von Gruppenleitungen der
17 // Tastatur zu Pins an Port A bzw. B:
18 // Port A
19 #define WIRE_PURPLE 0
```

```c
20 #define WIRE_ORANGE 1
21 // Port B
22 #define WIRE_GREEN 0
23 #define WIRE_RED 1
24 #define WIRE_BLUE 2
25 #define WIRE_BROWN 3
26 #define WIRE_YELLOW 4
27 #define WIRE_WHITE 6
28
29 // Definition von Tastencodes
30 #define KEY_0 0x00
31 #define KEY_1 0x01
32 #define KEY_2 0x02
33 #define KEY_3 0x03
34 #define KEY_4 0x04
35 #define KEY_5 0x05
36 #define KEY_6 0x06
37 #define KEY_7 0x07
38 #define KEY_8 0x08
39 #define KEY_9 0x09
40 #define KEY_A 0x0a
41 #define KEY_B 0x0b
42 #define KEY_C 0x0c
43 #define KEY_D 0x0d
44 #define KEY_E 0x0e
45 #define KEY_F 0x0f
46 #define KEY_LEFT 0x10
47 #define KEY_RIGHT 0x11
48 #define KEY_DOWN 0x12
49 #define KEY_UP 0x13
50 #define KEY_SK1 0x14
51 #define KEY_SK2 0x15
52 #define KEY_SK3 0x16
53 #define KEY_SK4 0x17
54 #define KEY_OK 0x18
55 #define KEY_PDOWN 0x19
56 #define KEY_PUP 0x1a
57 #define KEY_NONE 0xff
58
59 #endif
```

A.2.2 tastencontroller.c

```c
1 #include "tastencontroller.h"
2
3 // Maske fuer die Tastenkabel, die an Port A liegen.
4 const uint8_t WIRES_AT_A = (1<<WIRE_ORANGE) | (1<<WIRE_PURPLE);
5
6 // Maske fuer die Tastenkabel, die an Port B liegen.
7 const uint8_t WIRES_AT_B = (1<<WIRE_GREEN) | (1<<WIRE_RED)
```

```
8          | (1<<WIRE_BLUE) | (1<<WIRE_BROWN) | (1<<WIRE_YELLOW)
9          | (1<<WIRE_WHITE);
10
11 // Maske fuer die Tastengruppen, die an Port D liegen.
12 const uint8_t WIRES_AT_D = (1<<WIRE_GROUP1) | (1<<WIRE_GROUP2)
13         | (1<<WIRE_GROUP3) | (1<<WIRE_GROUP4);
14
15 /*
16  * Array der Gruppenleitungen, um besser ueber diese iterieren
17  * zu koennen.
18  */
19 const uint8_t groups[] =
20     {WIRE_GROUP1, WIRE_GROUP2, WIRE_GROUP3, WIRE_GROUP4};
21
22 /*
23  * Array der Tasten, um besser ablesen zu koennen, welche
24  * gedrueckt wurde.
25  */
26 const uint8_t keys[] = {KEY_0, KEY_1, KEY_2, KEY_3, KEY_4,
27         KEY_5, KEY_6, KEY_7, KEY_8, KEY_9, KEY_A, KEY_B, KEY_C,
28         KEY_D, KEY_E, KEY_F, KEY_LEFT, KEY_RIGHT, KEY_DOWN,
29         KEY_UP, KEY_SK1, KEY_SK2, KEY_SK3, KEY_SK4, KEY_OK,
30         KEY_PDOWN, KEY_PUP, KEY_NONE, KEY_NONE, KEY_NONE,
31         KEY_NONE, KEY_NONE};
32
33 // Diese Taste wurde gedrueckt.
34 uint8_t pressed_key = KEY_NONE;
35
36 // Diese Taste wurde auch wieder losgelassen.
37 uint8_t typed_key = KEY_NONE;
38
39 /*
40  * Prueft die uebergebene Gruppe auf eine gedrueckte Taste und
41  * gibt die Nummer der Gruppenleitung zurueck.
42  */
43 uint8_t check_group(uint8_t wire_group) {
44     // 0 an die gegebene Gruppe legen
45     DDRD |= (1<<wire_group);
46     // Stabilisierung abwarten
47     _delay_us(10);
48     // Speichern, an welchen Gruppenleitungen die 0 ankommt
49     uint8_t b = PINB | ~(WIRES_AT_B);
50     uint8_t a = PINA | ~(WIRES_AT_A);
51     // 0 wieder wegnehmen
52     DDRD &= ~(WIRES_AT_D);
53
54     if (b < 0xff || a < 0xff) {
55             // An einer der Gruppenleitungen kam die 0 an.
56             // Rueckgabewerte entsprechend der Gruppenleitung:
```

```
57          if ((b & (1<<WIRE_GREEN)) == 0x00) {
58              return 1;
59          } else if ((b & (1<<WIRE_RED)) == 0x00) {
60              return 2;
61          } else if ((b & (1<<WIRE_BLUE)) == 0x00) {
62              return 3;
63          } else if ((b & (1<<WIRE_BROWN)) == 0x00) {
64              return 4;
65          } else if ((b & (1<<WIRE_YELLOW)) == 0x00) {
66              return 5;
67          } else if ((b & (1<<WIRE_WHITE)) == 0x00) {
68              return 6;
69          } else if ((a & (1<<WIRE_PURPLE)) == 0x00) {
70              return 7;
71          } else if ((a & (1<<WIRE_ORANGE)) == 0x00) {
72              return 8;
73          }
74      }
75
76      // Keine Taste wurde gedrueckt
77      return 0;
78  }
79
80  /*
81   * Prueft alle Tastengruppen auf eine gedrueckte Taste und gibt
82   * diese zurueck.
83   */
84  uint8_t check_keystroke(void) {
85      // Nummer des Kabels der gedrueckten Taste
86      uint8_t wire;
87      // Nummer der aktuellen Tastengruppe
88      uint8_t group;
89
90      // Alle Gruppen auf eine gedrueckte Taste ueberpruefen,
91      // vorzeitig abbrechen, sobald eine gefunden wurde
92      for (group = 0; group < 4; group++) {
93          wire = check_group(groups[group]);
94
95          if (wire != 0) { // Eine Taste wurde gedrueckt
96              break;
97          }
98      }
99
100     if (wire != 0) { // Eine Taste wurde gedrueckt
101
102         // Die zu der Gruppe und dem Kabel gehoerende Taste
103         // zurueckgeben
104         return keys[group * 8 + wire - 1];
105     } else { // Keine Taste wurde gedrueckt
```

```
106             return KEY_NONE;
107         }
108 }
109 
110 /*
111  * Initialisiert den Tastencontroller.
112 Setzt Datenrichtungen
113  * und Pull-Up-Widerstaende der Ports und initialisiert USI.
114  */
115 void init(void) {
116     // Initialisiert USI
117     usi_init();
118 
119     // Setzt die Datenrichtungen fuer die Zeilen- und
120     // Spaltenleitungen und setzt sie auf lesend. Beim
121     // Ueberpruefen auf Tastendruecke wird Port D teilweise
122     // kurzzeitig auf schreibend gesetzt.
123     DDRA &= ~(WIRES_AT_A);
124     DDRB &= ~(WIRES_AT_B);
125     DDRD &= ~(WIRES_AT_D);
126 
127     // Aktiviert die Pull-Up-Widerstaende an den Tastenkabeln
128     // und deaktiviert sie fuer die Tastengruppen.
129     PORTA |= WIRES_AT_A;
130     PORTB |= WIRES_AT_B;
131     PORTD &= ~(WIRES_AT_D);
132 }
133 
134 /*
135     Wird beim Starten des Controllers aufgerufen. Initialisiert
136     den Controller und bearbeitet dann in einer Endlos-
137     schleife Tastendruecke und Anfragen des Hauptcontrollers.
138 */
139 int main(void) {
140     // Initialisierung
141     init();
142 
143     uint8_t usi_process_return;
144     uint8_t i = 0;
145     uint8_t j = 0;
146 
147     // Endlosschleife
148     while (1) {
149 
150         if (pressed_key == KEY_NONE) {
151 
152             // Noch keine Taste gedrueckt,
153             // Auf Tastendruck ueberpruefen
154             pressed_key = check_keystroke();
```

```
155             // Hilfsvariable zuruecksetzen
156             i = 0;
157 
158         } else if (typed_key == KEY_NONE) {
159             // Taste wurde noch nicht losgelassen
160 
161             // ~20 ms warten (Entprellung), dazwischen USI
162             // bearbeiten (Fehlendes Delay passiert in
163             // usi_process()
164             if (i < 2) {
165                 _delay_ms(1);
166                 i++;
167             } else if (check_keystroke() == KEY_NONE) {
168                 // Taste wurde losgelassen
169                 typed_key = pressed_key;
170             }
171         }
172 
173         // USI bearbeiten und Rueckgabe speichern
174         usi_process_return = usi_process();
175 
176         if (usi_process_return == USI_NOP) {
177             // Nichts interessantes passiert
178 
179             // Timeout-Zaehler erhoehen
180             j++;
181 
182             if (j > USI_TIMEOUT) {
183                 // Timeout erreicht, Fehler annehmen
184 
185                 // USI zuruecksetzen
186                 j = 0;
187                 usi_reset();
188             }
189 
190         } else { // Etwas interessantes passiert
191             // Timeout zuruecksetzen
192             j = 0;
193 
194             if (usi_process_return == USI_SET_DATA) {
195                     // Taste muss an USI gegeben werden
196 
197                 // Gedrueckte Taste uebergeben
198                 usi_set_data(typed_key);
199 
200                 if (typed_key != KEY_NONE) {
201                         // Eine Taste war getippt worden
202 
203                     // Tastendruecke zuruecksetzen
```

```
204                         pressed_key = KEY_NONE;
205                         typed_key = KEY_NONE;
206                     }
207
208                 } else if (usi_process_return == USI_GET_DATA) {
209                     // Ein Datum wurde empfangen -- Fehler
210                     // annehmen, da der Tastencontroller nur
211                     // sendet.
212                     usi_reset();
213                 }
214             }
215         }
216         return 0;
217 }
```

A.2.3 usi.h

```
1  #ifndef __USI_H__
2  #define __USI_H__
3
4  #include <avr/io.h>
5  #include <inttypes.h>
6  #include <util/delay.h>
7
8  // Konstanten, die fuer die Pins von SDA und SCL stehen.
9  #define SDA DDB5
10 #define SCL DDB7
11
12 // Konstanten, die die eigene Adresse und das Timeout angeben.
13 #define USI_ADDRESS 2
14 #define USI_TIMEOUT 100
15
16 // Konstanten, die fuer die Zustaende stehen.
17 #define USI_STATE_START 0
18 #define USI_STATE_ADDRESS 1
19 #define USI_STATE_READ_ACK 2
20 #define USI_STATE_READ_SET 3
21 #define USI_STATE_READ 4
22 #define USI_STATE_WRITE_ACK 5
23 #define USI_STATE_WRITE 6
24 #define USI_STATE_WRITE_GET 7
25
26 // Konstanten, die fuer Statusmeldunden stehen.
27 #define USI_NOP 0
28 #define USI_SET_DATA 1
29 #define USI_GET_DATA 2
30
31 void usi_init(void);
32
33 void usi_set_data(uint8_t data);
```

```
34
35 uint8_t usi_get_data(void);
36
37 uint8_t usi_process(void);
38
39 void usi_reset(void);
40
41 #endif
```

A.2.4 usi.c

```
1  #include "usi.h"
2
3  // Speichert den aktuellen Zustand.
4  uint8_t usi_state = USI_STATE_START;
5
6  // Speichert den zuletzt uebertragenen Wert
7  uint8_t usi_data;
8
9  /*
10  * Initialisiert USI.
11 Aktiviert USI und konfiguriert
12  * SDA und SCL.
13  */
14 void usi_init(void) {
15     USICR = (1<<USIWM1) | (1<<USIWM0) | (1<<USICS1);
16     PORTB |= (1<<SDA) | (1<<SCL);
17     DDRB &= ~(1<<SDA);
18     DDRB |= (1<<SCL);
19 }
20
21 /*
22  * Gibt zurueck, ob Startcondition empfangen wurde.
23  */
24 uint8_t usi_is_start(void) {
25     return (USISR & (1<<USISIF));
26 }
27
28 /*
29  * Gibt zurueck, ob Overflow passiert ist. Es wurden also 8 Bit
30  * empfangen.
31  */
32 uint8_t usi_is_overflow(void) {
33     return (USISR & (1<<USIOIF));
34 }
35
36 /*
37  * Gibt zurueck, ob Stopcondition empfangen wurde.
38  */
39 uint8_t usi_is_stop(void) {
```

```
40      return (USISR & (1<<USIPF));
41  }
42
43  /*
44   * Gibt zurueck, ob eine Kollision bemerkt wurde.
45   */
46  uint8_t usi_is_collision(void) {
47      return (USISR & (1<<USIDC));
48  }
49
50  /*
51   * Setzt den zu uebertragenen Wert und sendet diesen.
52   */
53  void usi_set_data(uint8_t data) {
54
55      // Nur senden, wenn im richtigen Zustand.
56      if (usi_state == USI_STATE_READ_SET) {
57          usi_state = USI_STATE_READ;
58          USIDR = data;
59          USISR |= (1<<USIOIF);
60      }
61  }
62
63  /*
64   * Gibt den zuletzt empfangenen Wert zurueck.
65   */
66  uint8_t usi_get_data(void) {
67
68      // Nur bestaetigen, wenn im richtigen Zustand.
69      if (usi_state == USI_STATE_WRITE_GET) {
70          usi_state = USI_STATE_WRITE_ACK;
71          USISR |= (1<<USIOIF);
72      }
73
74      return usi_data;
75  }
76
77  /*
78   * Prueft, ob sich bei der Uebertragung etwas getan hat und
79   * reagiert darauf.
80   */
81  uint8_t usi_process(void) {
82      // Warten, um doppelte Abarbeitung desselben Events zu
83      // verhindern
84      _delay_ms(8);
85
86      if (usi_is_start()) { // Startcondition
87          usi_state = USI_STATE_ADDRESS;
88          USISR = (1<<USISIF);
```

```
89      } else if (usi_is_overflow()) { // Overflow
90
91          // Abhaengig vom Zustand reagieren
92          switch (usi_state) {
93
94              // Adresse wurde uebertragen
95              case USI_STATE_ADDRESS:
96
97                  if ((USIDR>>1) == USI_ADDRESS) {
98                          // Eigene Adresse wurde angesprochen
99
100                     if (USIDR & 0x01) {
101                             // Wert soll geschrieben werden
102                         usi_state = USI_STATE_READ_ACK;
103                     } else {
104                             // Wert soll gelesen werden
105                         usi_state = USI_STATE_WRITE_ACK;
106                     }
107
108                     // ACK senden
109                     DDRB |= (1<<SDA);
110                     USIDR = 0x00;
111                     USISR = 0x0e | (1<<USIOIF);
112                 } else { // Im falschen Zustand
113                     usi_reset();
114                 }
115
116                 break;
117
118             // ACK wurde gesendet, Wert soll nun gesetzt werden
119             case USI_STATE_READ_ACK:
120
121                 if (USIDR & 0x01) { // Kein ACK
122                     usi_reset();
123                 } else { // ACK
124
125                     // Wert muss nun gesetzt werden
126                     usi_state = USI_STATE_READ_SET;
127                     DDRB |= (1<<SDA);
128                     return USI_SET_DATA;
129                 }
130
131                 break;
132
133             // Wert muss immer noch gesetzt werden
134             case USI_STATE_READ_SET:
135                 return USI_SET_DATA;
136                 break;
137
```

```
138             // ACK kann empfangen werden
139             case USI_STATE_READ:
140                 usi_state = USI_STATE_READ_ACK;
141                 DDRB &= ~(1<<SDA);
142                 USISR = 0x0e;
143                 USISR = (1<<USIOIF);
144                 break;
145
146             // ACK wurde gesendet, Wert kann empfangen werden
147             case USI_STATE_WRITE_ACK:
148                 usi_state = USI_STATE_WRITE;
149                 DDRB &= ~(1<<SDA);
150                 USISR = (1<<USIOIF);
151                 break;
152
153             // Wert wurde empfangen, ACK vorbereiten,
154             //  Wert muss erst abgeholt werden
155             case USI_STATE_WRITE:
156                 usi_state = USI_STATE_WRITE_GET;
157                 usi_data = USIDR;
158                 DDRB |= (1<<SDA);
159                 USIDR = 0x01;
160                 USISR = 0x0e;
161                 return USI_GET_DATA;
162                 break;
163
164             // Wert muss immer noch abgeholt werden
165             case USI_STATE_WRITE_GET:
166                 return USI_GET_DATA;
167                 break;
168
169             default: // Unbehandelter Fall, Fehler annehmen
170                 usi_reset();
171                 break;
172
173         }
174
175     } else if (usi_is_stop()) { // Stopcondition
176         // Zuruecksetzen
177         usi_reset();
178     } else if (usi_is_collision()) { // Kollision
179         // Zuruecksetzen
180         usi_reset();
181     }
182
183     // Nichts interessantes passiert
184     return USI_NOP;
185 }
186
```

```
187 /*
188  * USI zuruecksetzen.
189  */
190 void usi_reset(void) {
191     usi_state = USI_STATE_START;
192     DDRB &= ~(1<<SDA);
193     USISR = (1<<USISIF) | (1<<USIOIF) | (1<<USIPF) | (1<<USIDC);
194 }
```

B Bedienungsanleitung

Der EPROM-Programmer wird über die beiden Laborbuchsen auf der Rückseite mit Spannung versorgt. Nach dem Anlegen von 5 V Gleichspannung ist das Gerät sofort betriebsbereit, und im Display wird das Hauptmenü angezeigt.

Die Menüpunkte, die in der untersten Zeile angezeigt werden, können durch Drücken des Softkeys direkt darunter aufgerufen werden.

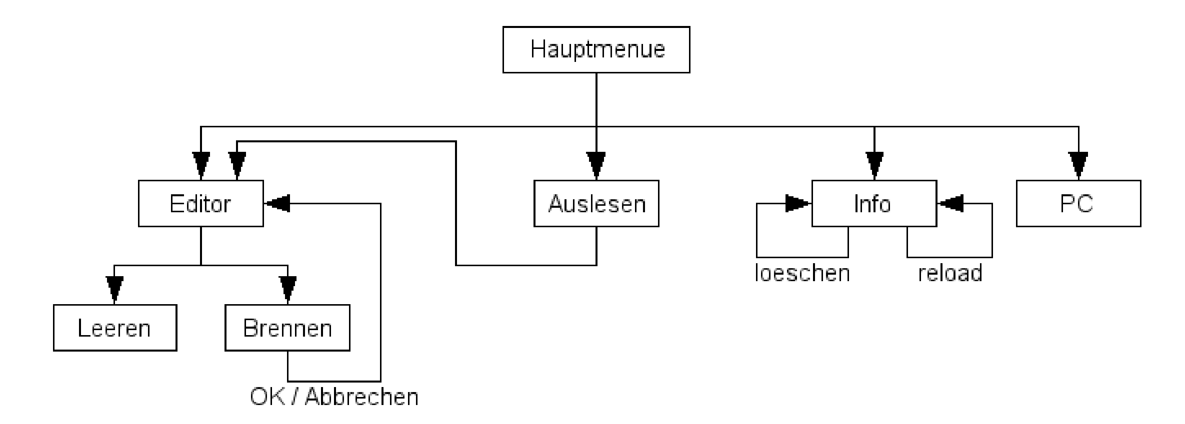


Abbildung 11: Das Menüsystem des Geräts. Von allen Untermenüs aus kann über den rechten Softkey (dann mit „zurück“ beschriftet) ins nächsthöhere Menü zurückgewechselt werden.

B.1 Informationen über eingelegte PROMs abrufen

Die Funktion „Info“ im Hauptmenü liefert Informationen über den eingelegten Speicherbaustein. Diese Funktion ermöglicht eine schnelle Prüfung, ob ein PROM leer ist.

Es wird angezeigt, ob ein EPROM oder ein EEPROM eingelegt ist. Dies wird aber nur anhand der Schalterstellung des Schalters S1 erkannt, so dass das erkannte PROM nicht dem eingelegten entsprechen muss. Da ein EEPROM durch eine zu hohe Brennspannung kaputt gehen kann, ist dringlich darauf zu achten, dass Schalterstellung und eingelegtes PROM zueinander passen.

Ebenfalls wird angezeigt, ob das eingelegte PROM leer ist oder Daten enthält. Dabei wird angenommen, dass ein PROM leer ist, wenn alle Bytes den Wert 0xFF haben.

Ist ein nicht-leeres EEPROM eingelegt, so kann über „loeschen“ das EEPROM gelöscht werden, es werden dabei alle Bytes mit dem Wert 0xFF beschrieben.

Über die Funktion „reload“ können die Informationen aktualisiert werden. Dies ist nützlich, wenn mehrere PROMs auf Inhalt überprüft werden sollen.

B.2 Daten eingeben und brennen

Ein Druck auf den Softkey „Editor“ startet den Editor.

In der Editoransicht steht links in jeder Zeile die Anfangsadresse des ersten Bytes, das in dieser Zeile angezeigt wird, in hexadezimaler Schreibweise. Nach einem Doppelpunkt folgen 8 Bytes Daten, ebenfalls in hexadezimaler Schreibweise.

Der Cursor, als Unterstrich dargestellt, steht beim ersten Start des Editors auf Position 000h. Wird der Editor verlassen (z.B. über die Funktion„zurück“ zum Hauptmenü) und wieder aufgerufen, bleibt die letzte Cursorposition und alle eingegebenen Daten erhalten.

Daten lassen sich über die Tastatur direkt eingeben; der Cursor kann mit den Cursortasten frei verschoben werden. Die Tasten PgUp und PgDn blättern seitenweise durch den Editorpuffer. Fehler können durch einfaches Überschreiben der falsch eingegebenen Bytes korrigiert werden; die Funktion „Leeren“ leert auf Wunsch den gesamten Editorpuffer.

Sind die gewünschten Daten komplett eingegeben, werden sie nach Aufruf des Menüpunkts „Brennen“ und anschließender Bestätigung ins EPROM geschrieben.

B.3 PROMs auslesen und kopieren

Die Funktion „Auslesen“ im Hauptmenü liest die Daten eines PROMs, das in der Fassung des Programmiergeräts steckt, aus und kopiert sie in den Editor. Anschließend wird der Editor angezeigt. Damit lassen sich einerseits Daten kontrollieren, die in PROMs gespeichert sind, andererseits kann diese Funktion auch zum einfachen Kopieren von PROMs genutzt werden: Nachdem ein PROM mit den gewünschten Daten ausgelesen wurde, können diese aus dem Editor in beliebig viele weitere PROMs gebrannt werden.

B.4 PC-Steuerung

Über die Funktion „PC“ wird das Gerät für die PC-Steuerung vorbereitet. Nach dem Aufruf dieser Funktion kann vom PC-Programm aus die Verbindung zum Programmiergerät hergestellt werden. Es kann so vom PC aus gesteuert werden.

C Schaltpläne

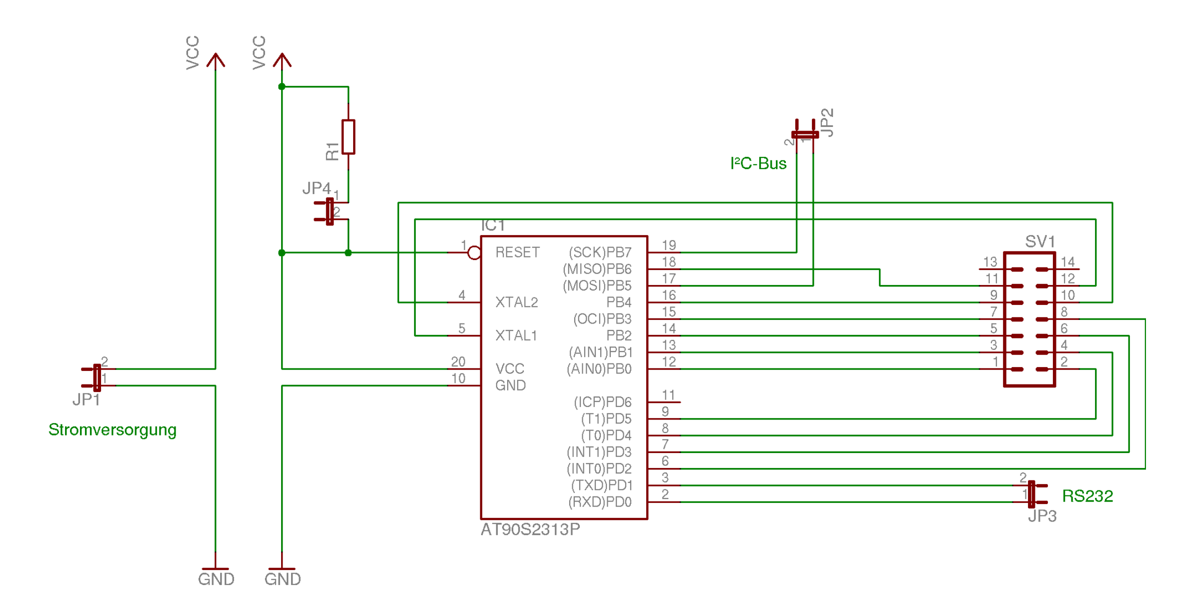


Abbildung 12: Schaltplan des Tastaturcontrollers. An SV1 sind die Zeilen- und Spaltenleitungen der Tastaturmatrix angeschlossen.

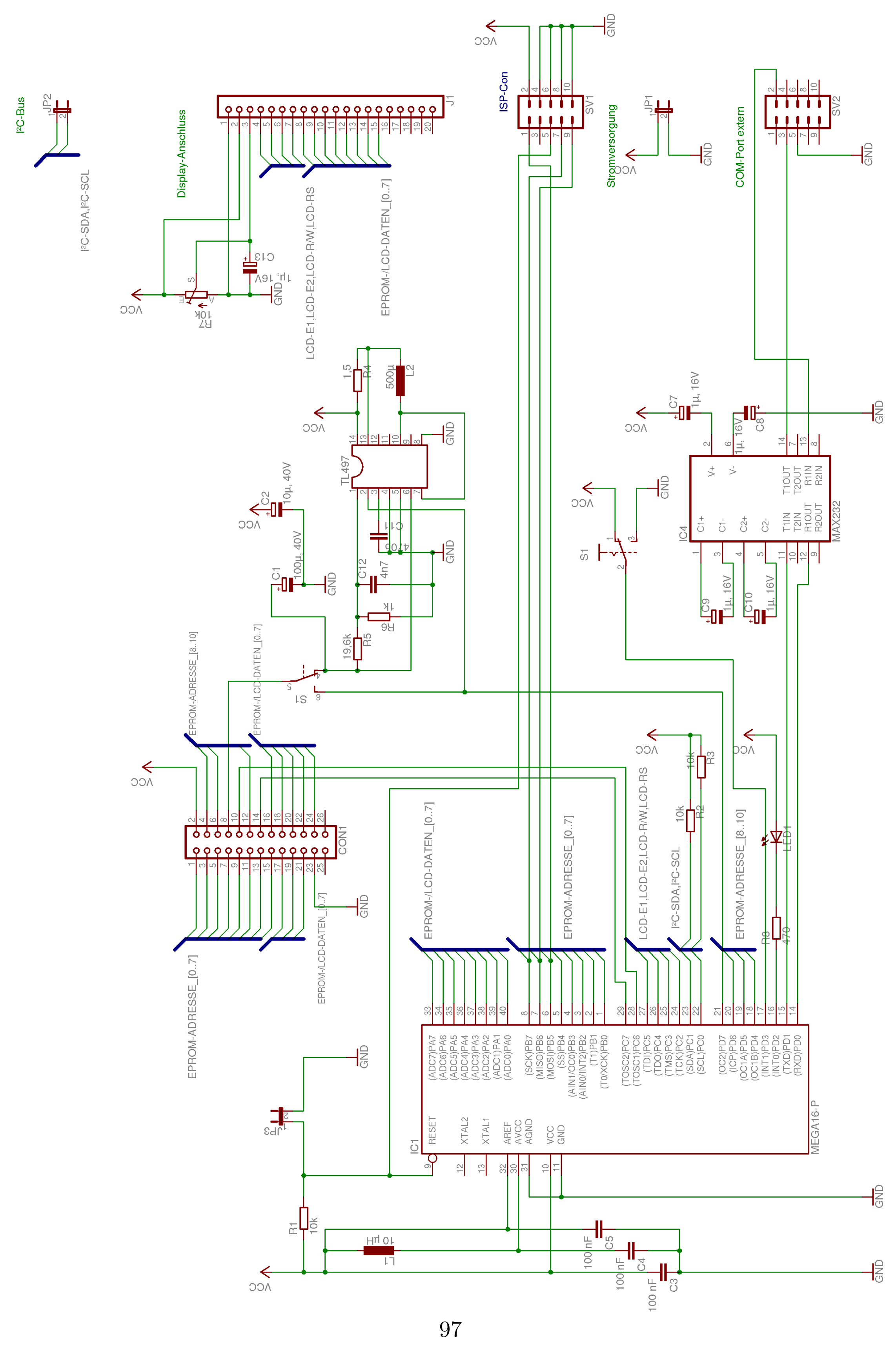


Abbildung 13: Schaltplan der Hauptplatine

D Platinenlayouts

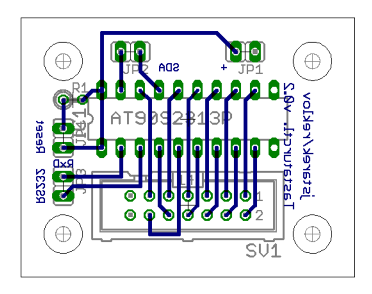


Abbildung 14: Layout der Platine des Tastaturcontrollers

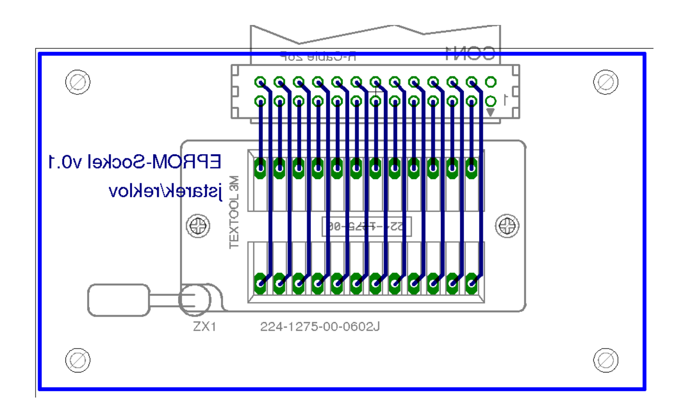


Abbildung 15: Layout der Platine für den abgesetzten IC-Sockel

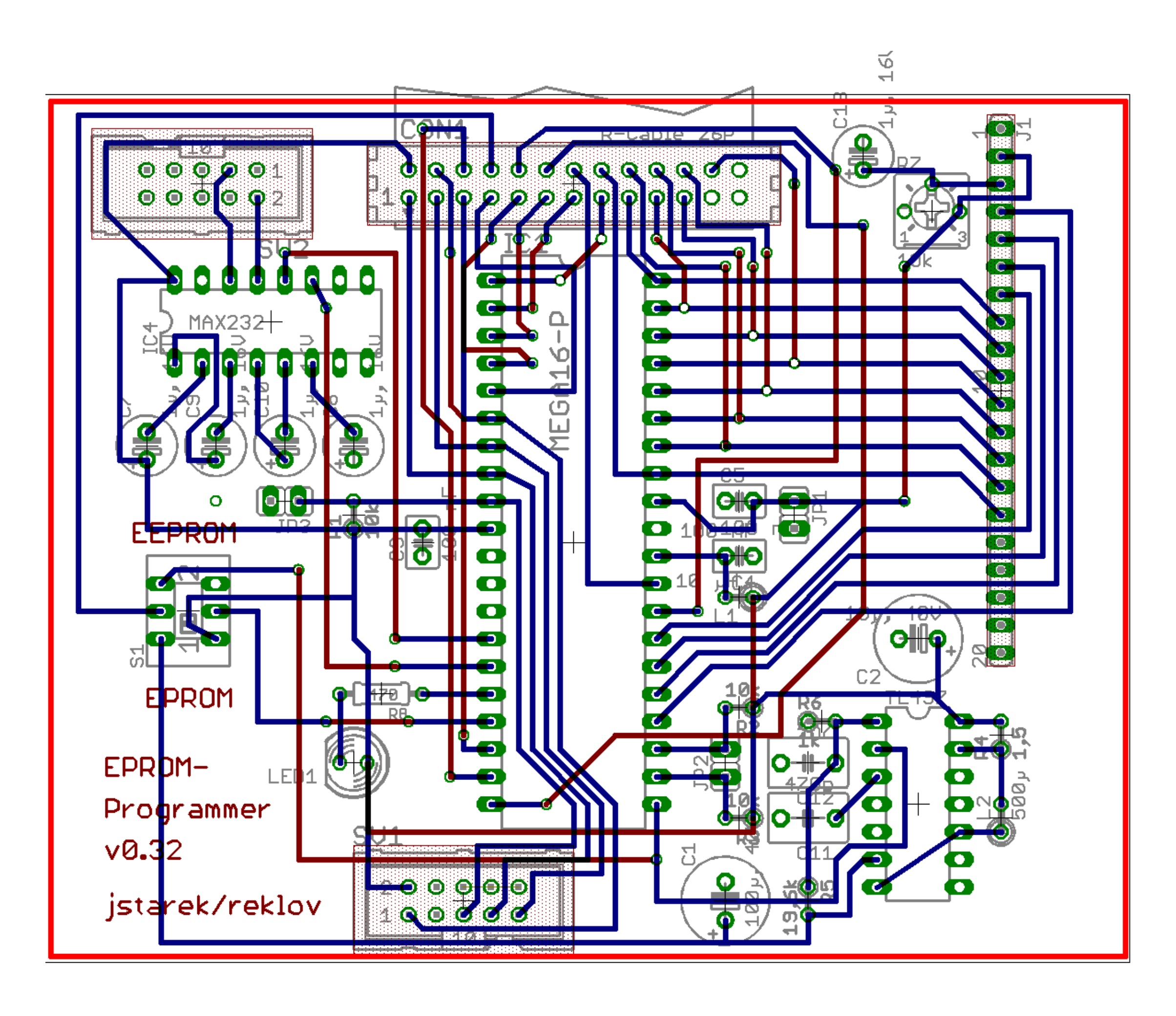


Abbildung 16: Layout der Hauptplatine, Version 0.32. Blaue Leiterbahnen verlaufen auf der Löt-, rote auf der Bestückungsseite.

Literatur

[SIE88] Siemens AG (1988): „Experimentiercomputer ECB85. Bedienungsanleitung 11.81“. München.

[FA89] „Funkamateur Bauteileinformation U 6516 DG“. In: Funkamateur 5/89, S. 233 ff.

[Hit44780] Hitachi (ohne Jahr): „HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver“. Ohne Ort.

[KSW06] Kories, Ralf und Schmidt-Walter, Heinz (2006): „Taschenbuch der Elektrotechnik“. Frankfurt: Verlag Harry Deutsch.

[MAX04] Maxim Integrated Products (2004): „MAX220-MAX249. +5V-Powered, Multichannel RS-232 Drivers/Receiver“. Rev. 14.

[Tho94] SGS-Thomson Microelectronics (1994): „M2716 data sheet“. Ohne Ort.

[TI95] Texas Instruments Inc. (1995): „TL497AC, TL497AI, TL497AC Switching Voltage Regulators“. Dallas.

Die im Text angegebenen URLs wurden am 17. Juli 2007 abgerufen.